

TOWARD ROBUST HIGH PERFORMANCE DISTRIBUTED SERVICES

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Yee Jiun Song

August 2011

© 2011 Yee Jiun Song
ALL RIGHTS RESERVED

TOWARD ROBUST HIGH PERFORMANCE DISTRIBUTED SERVICES

Yee Jiun Song, Ph.D.

Cornell University 2011

This thesis presents steps towards simplifying the implementation of robust high performance distributed services.

First, we investigate consensus algorithms in the context of fault tolerant systems. Consensus algorithms, often a critical part of fault tolerant systems, are notoriously difficult to implement. We present a skeleton consensus algorithm that can be instantiated into several well-known consensus protocols, providing insight into the structure of consensus algorithms as well as the differences and performance tradeoffs between different algorithms. We investigate one-step Byzantine agreement algorithms which exploit contention-free situations to provide low latency performance. We present definitions of one-step algorithms and prove a lower bound on the number of processors required for such algorithms, thereby showing that our algorithm is optimal. We then generalize our investigation to k -set agreement.

Second, we present RPC chains, a communication primitive that improves the performance of geodistributed enterprise applications. Distributed enterprise applications are often built as a composition of more basic services. Currently, such compositions are built using remote procedure calls (RPCs). This results in a rigid and inefficient communication pattern. RPC chains is a new abstraction that applies two well-known ideas, function shipping and continuations, to allow an improvement in the latency performance and reduction in the overall bandwidth usage of such geodistributed systems.

BIOGRAPHICAL SKETCH

Yee Jiun Song was born September 16, 1979 in Johor Bahru, Malaysia. He graduated from the University of California, Berkeley with a Bachelor of Science in Electrical Engineering and Computer Sciences and a Bachelor of Arts in Economics in 2002. He then earned a Master of Science in Computer Science at Stanford University before beginning his Ph.D. studies at Cornell University.

In loving memory of my father.

ACKNOWLEDGEMENTS

I am most grateful to my parents for making every opportunity available to me throughout the years, and encouraging curiosity, hard work, and the pursuit of excellence. I am especially thankful to my mother for being a pillar of strength after my dad's unexpected passing, and for doing everything in her power to ensure that my sister and I continued to have everything we needed to complete our education without any disruption.

I am fortunate to have had many great mentors and colleagues throughout the years. I would like to thank my high school teachers Mr. R. Ilango and Mr. Harphal Singh, for sparking my interest in programming and teaching me how to think independently. At Cornell, I am grateful to have worked with a number of brilliant faculty members. In particular, Ken Birman was a constant and generous source of reliable advice. I also benefitted greatly from all the discussions I have had with my fellow graduate students, in particular, I am grateful for the many hours of conversations I have had with my officemates Jonathan Kaldor and Ian Kash. I am also grateful to the colleagues that I met during my internships at Microsoft Research and Yahoo! Research. Most of all, I am grateful to my advisor, Robbert van Renesse, for his patience and guidance over the last few years. I could not have asked for a better advisor, and would likely have not completed my PhD if Robbert had not picked me up as a student. Robbert has been a role model and an inspiration, and much of what I know about research and being a scientist, I learned from him.

Lastly, I would like to thank Ling for all her patience and support.

TABLE OF CONTENTS

Biographical Sketch	iii
Dedication	iv
Acknowledgements	v
Table of Contents	vi
List of Tables	viii
List of Figures	ix
1 Introduction	1
1.1 Fault Tolerance Made Easy	2
1.2 Improving The Performance Of Geodistributed Applications . . .	4
1.3 Related Work and Background	5
1.3.1 Consensus and Fault Tolerance	5
1.3.2 Function Shipping and Continuations	8
2 The Building Blocks of Consensus	12
2.1 The Consensus Problem	12
2.2 Extended Quorum Systems	15
2.3 The Consensus Skeleton	19
2.3.1 Instances	20
2.3.2 Guarded Proposal	22
2.3.3 Extended Quorum System API	24
2.3.4 State Maintenance	25
2.3.5 The Instance Mechanism	26
2.4 Full Protocols	34
2.4.1 Paxos	34
2.4.2 Chandra-Toueg	35
2.4.3 Ben-Or	37
2.5 Implementation and Protocol Comparisons	38
2.5.1 Implementation	39
2.5.2 Experimental Setup	41
2.5.3 Results	41
3 Bosco: One-Step Byzantine Asynchronous Consensus	48
3.1 The Byzantine Consensus Problem	49
3.2 Lower Bounds	52
3.2.1 Lower Bound for Strongly One-Step Byzantine Consensus	52
3.2.2 Lower Bound for Weakly One-Step Byzantine Consensus .	55
3.3 Bosco	56
3.4 RS-Bosco and WS-Bosco	61

4	<i>k</i>-Set Agreement in One Communication Step	70
4.1	<i>k</i> -Set Agreement	71
4.2	Lower Bounds	72
4.3	Algorithm	78
5	RPC Chains: Efficient Client-Server Communication in Geodistributed Systems	84
5.1	Setting	87
5.2	Design	88
5.2.1	Main mechanism	88
5.2.2	Chaining function repository	90
5.2.3	Parameters and state	91
5.2.4	Nesting and composition	91
5.2.5	Support for legacy RPC services	93
5.2.6	Isolation	94
5.2.7	Debugging and profiling	95
5.2.8	Exceptions	96
5.2.9	Broken chains	97
5.2.10	Splitting chains	98
5.3	Applications	100
5.3.1	Storage applications	100
5.3.2	Web mail application	102
5.4	Evaluation	104
5.4.1	Setup	105
5.4.2	Microbenchmarks	105
5.4.3	Storage application	109
5.4.4	Web mail application	113
5.5	Limitations	115
6	Conclusion and Future Work	117
	Bibliography	122

LIST OF TABLES

2.1	Terminology.	16
2.2	Size requirements for Threshold Quorum Systems that satisfy consistency and opaqueness.	18

LIST OF FIGURES

2.1	The actions of the various actors.	20
2.2	An instance of the consensus protocol.	27
2.3	Mean time to decide a single value under varying request rates .	42
2.4	Median time to decide a single value under varying request rates	43
2.5	Communication overhead under varying request rates	44
2.6	Mean time to decide a single value under varying failure rates .	45
2.7	Median time to decide a single value under varying failure rates	45
2.8	Communication overhead under varying failure rates	46
5.1	Standard RPCs vs an RPC chain	85
5.2	Signature of a service function and chaining function; signature of a function that initiates an RPC chain.	89
5.3	Execution of an RPC chain.	89
5.4	Composition of nested chains.	92
5.5	Copying data between two storage servers using RPCs, RPC chains, and RPC chains with composition.	101
5.6	A simplified web mail server that uses RPC chains.	103
5.7	Latency and bandwidth between pairs of sites.	105
5.8	Overhead for compiling chaining functions and storing com- piled code.	106
5.9	Executions used in the experiment of Section 5.4.2.	107
5.10	Execution time using an RPC chain versus standard RPC to call 2 servers.	108
5.11	Comparison of RPC copy and Chain copy under various set- tings: clients collocated with servers in mountain view; client in redmond; and client in Beijing.	109
5.12	Throughput-latency of RPC copy and Chain copy.	111
5.13	Benefit of chain composition.	112
5.14	RPC chain in web mail application.	113

CHAPTER 1

INTRODUCTION

Distributed systems have never been more ubiquitous. Many of the services that people depend on on a daily basis, ranging from email and various web services to the stock exchange and even critical services such as air traffic control are based on distributed systems. As a result, much of systems research over the last decade has focused on distributed systems, with a particular emphasis on large scale systems that live in the datacenter. Yet despite all the advancements in distributed systems technology, building robust distributed systems remain one of the most daunting tasks that a software engineer can be faced with.

Broadly speaking, the work in this dissertation aims at providing tools that make it easier to build robust, high performance distributed systems. First, we present research in consensus protocols aimed at simplifying the building of high performance fault tolerant systems; second, we present RPC chains, a communication primitive that improves the communication pattern of geodistributed applications, enabling better performance and lower bandwidth usage.

Overall, our main contributions are as follows:

- a skeleton consensus algorithm that can be instantiated into several well-known consensus protocols, demonstrating that they require not one, but two separate quorum systems;
- definitions, lower bounds, and an optimal algorithm for *one-step* Byzantine agreement, which optimizes performance in contention-free situations;
- a generalization of one-step algorithms to k -set agreement;

- RPC chains, a communication primitive that applies concepts of function shipping and continuations to improve the performance of geographically distributed enterprise applications.

1.1 Fault Tolerance Made Easy

Computers will fail, and for many systems it is imperative that such failures be tolerated. State machine replication is a general approach for providing fault tolerance [64]. Roughly speaking, systems can be engineered to tolerate failures by having multiple *replicas* and using a protocol to keep the replicas in sync. When a failure affects one or more of the replicas, the other replicas can be depended upon to provide availability. While this approach seems simple in theory, in practice, building a fault tolerance system can be very challenging. In particular, consensus protocols, which are used to keep replicas synchronized, are notoriously difficult to understand and to implement [18, 40]. It is thus desirable to avoid having to re-implement consensus unless absolutely necessary.

One way to avoid having developers implement consensus from scratch every time a fault tolerant system is built is to encapsulate the functionality of consensus into a toolkit, and provide developers with a simple abstraction that can be used to build the mission critical parts of their application. Examples of such an approach are Chubby [14] and ZooKeeper [1]. While such systems do simplify the task of building a fault tolerant system, they are still less than ideal. They force developers to use a rigid abstraction (often that of a file system) that may or may not be suitable for the particular application. More often than not, developers are still required re-engineer their applications from scratch with

fault tolerance in mind, albeit avoiding the task of implementing consensus.

We believe that a simpler approach is possible. Given recent advances in virtual machine and compiler technology, it is not difficult to imagine a straight forward implementation of the state machine replication approach in the form of an automatic replication/fault tolerance platform. In such a system, the developer simply builds a standalone application and runs it on the replication platform. The replication platform would be responsible for running multiple copies of the application on a cluster of machines, and keep those replicas in sync using an appropriate consensus protocol. When a machine fails, the platform would simply start a new copy of the application on another machine, and transfer the appropriate state from one or more of the remaining healthy replicas. This would fully automate the implementation of fault tolerance; from the point of view of the application developer, building a fault tolerant application would then be no different than building a standalone application.

However, the above description ignores performance considerations. Much of the performance bottleneck of such a system would stem from the consensus protocol that is used. Since the original *agreement* or *consensus* problem was proposed in [61], many versions of the problem and corresponding solutions have been introduced (see [8] for a survey of just the first decade, containing well over 100 references). Different protocols have different performance characteristics under different kinds of workloads; selecting the right protocol is key to providing good performance for an automatic fault tolerance platform.

In Chapter 2, we look at different consensus protocols and propose a skeleton algorithm that highlights the commonalities and differences between several well-known consensus protocols and provide insight into how different

protocols perform under different circumstances [67]. In Chapter 3, we propose *One-Step* Byzantine agreement protocols that are optimized for workloads where contention is the exception rather than the rule [66]. Finally, in Chapter 4, we generalize this result for k -Set Agreement [21].

1.2 Improving The Performance Of Geodistributed Applications

Enterprise applications in datacenters are often a composition of many underlying services. Oftentimes, these services are not just distributed within a single data center but can instead be spread over multiple data centers. As these applications get more sophisticated, the communication pattern between underlying components that make up a distributed application becomes critical to the overall performance of the system. This results in a performance bottleneck because RPCs, which is the communication primitive most often used to connect these components, leads naturally to a communication pattern that is less than ideal. In Chapter 5, we propose a simple but powerful communication primitive, RPC Chains [65], that applies two well-known ideas, *function shipping* and *continuations*, to the context of geodistributed enterprise applications to allow an improvement in the latency performance and a reduction in overall bandwidth usage. In addition, we propose ways to provide developers with the tools to debug and isolate problems using RPC chains, and evaluate our design using two sample applications.

1.3 Related Work and Background

1.3.1 Consensus and Fault Tolerance

There is a considerable body of work on the understanding and unification of consensus protocols. In [56], Mostefaoui and Raynal present a generic quorum based consensus protocol that works with any failure detector in the class \mathcal{S} or the class $\diamond\mathcal{S}$. Guerraoui and Raynal [30] point out similarities between different consensus protocols. They provide a generic framework for consensus algorithms and show that differences between the various algorithms can be factored out into a function called Lambda . Each consensus algorithm employs rather different implementations of Lambda . Later, Guerraoui and Raynal [31] show that leader-based algorithms can be factored into an Omega module and an Alpha module, where all differences are captured by differences in Omega .

In [36], Hurfin et al. present a unifying approach for instantiating a large class of failure detector-based algorithms that use either a \mathcal{S} or $\diamond\mathcal{S}$ failure detector, and whose message pattern in each round vary from $O(n)$ to $O(n^2)$. In Hurfin et al.'s algorithm, the role of the *Deciders* is similar to the role of deciders we will present in this thesis, and the *Agreement Keepers* play a role similar to that of the registrars. However, Hurfin et al.'s framework is restricted to leader-based algorithms and does not have an explicit selector-equivalent.

Zielinski [78] presents a framework for expressing various consensus protocols using an abstraction called *Optimistically Terminating Consensus*(OTC). In [79], Zielinski uses the OTC framework to automatically discover and verify consensus protocols. Like the algorithms in [30, 31, 36], the class of algorithms

that can be instantiated with the OTC framework include only failure detector-based algorithms, and in particular, cannot be used to instantiate randomized algorithms such as the Ben-Or algorithm.

In [52], Milosevic et al. proposes an abstraction called weak interactive consistency that unifies Byzantine consensus algorithms with and without signed messages. This allows concise expressions of algorithms such as PBFT [16] and FaB Paxos [50].

Li et al. [46] propose the Paxos register which unifies Paxos-style consensus protocols. They use the Paxos register to demonstrate the similarities and differences between Paxos [42], PBFT [16], and FaB Paxos [50].

Guerraoui et al. [29] proposed an abstraction called *abstract* that allows BFT protocols to be described as a composition of abstracts, each of which is an abortable state machine. This provides modularity in the implementation of BFT protocols and significantly reduces implementation efforts.

Many techniques have been proposed to improve the performance and reduce the overhead of providing Byzantine fault tolerance. Abd-El-Malek et al. [2] proposed the optimistic use of quorums rather than agreement protocols to obtain higher throughput. However, in the face of contention, optimistic quorum systems perform poorly. HQ combines the use of quorums and consensus techniques to provide high performance during normal operation and minimize overhead during periods of contention [22]. Probabilistic techniques have also been proposed to reduce the overhead of using quorum systems to provide Byzantine fault-tolerance [51, 48]. Hendricks et al. [34] proposed the use of erasure coding to minimize the overhead of a Byzantine fault-tolerant

storage system. Zyzzyva uses optimistic speculation to decrease the latency observed by clients [41].

Lamport [44] presents lower bounds for the number of message delays and the number of processors needed for several kinds of asynchronous non-Byzantine consensus algorithm in; in particular, *Fast Learning* algorithms are one-step algorithms for non-Byzantine settings. A one-step version of Paxos [42], Fast Paxos, is presented in [11, 45]. Fast Paxos tolerates only crash failures, although [45] alludes to the possibility of a Byzantine fault-tolerant version of Fast Paxos.

Brasileiro et al. [13] proposed a general technique for converting any crash-tolerant consensus algorithm into a crash-tolerant consensus algorithm that terminates in one communication step if all correct processors have the same initial value. Bosco, presented in Chapter 3 is an extension of the ideas presented in that work to handle Byzantine failures. The key difference between handling crashed failures and Byzantine failures is that when Byzantine failures need to be tolerated, equivocation must be handled correctly.

A simple and elegant crash-tolerant consensus algorithm of the same flavor, One-Third-Rule, appears in [19]. This work has been extended to handle Byzantine faults by considering transmission faults where messages can be corrupted in addition to being dropped [10]. The algorithms in [19, 10] differ from the algorithms we present in this thesis because they use a different failure model, where failures are attributed to message transmissions, rather than to processors.

Friedman et al. [27] proposed a weakly one-step algorithm that tolerates

Byzantine faults and terminates with probability 1 but does not tolerate a strong network adversary. In particular, their protocol is dependent on a common coin oracle and assumes that the network adversary has no access to this common coin; a strong network adversary with access to the common coin can prevent termination.

Among the algorithms constructed in [78] are two Byzantine consensus algorithms with one-step characteristics that require $n > 5t$ and $n > 3t$. The first of these algorithms is a weakly one-step algorithm that requires partial synchrony; the second algorithm, while appearing to violate the lower bounds we will show in Chapter 3, is neither weakly nor strongly one-step because processors can only decide in the first communication step when, in addition to the system being failure-free and contention-free, all processors are fast enough that the timeout mechanism in the algorithm is not triggered.

1.3.2 Function Shipping and Continuations

RPC chains utilize two well-understood ideas in the context of remote execution: *function shipping* and *continuations*.

Function shipping is the general technique of sending computation to the data rather than bringing the data to the computation. It is used in some systems where the cost of moving data is large compared to the cost of moving computation. For example, Diamond [37] is a storage architecture in which applications download searchlet code to disk to perform efficient filtering of large data sets locally, thereby improving efficiency. RPC chains use function shipping to send chaining logic.

A continuation [69] refers to the shifting of program control and transfer of current state from one part of a program to another. Extending this to *distributed continuations* is a natural step, allowing a continuation to shift program control from one processor to another. Several works in the parallel programming community give high-level programming continuation constructs and specify their behavior formally, e.g., [53, 38]. Distributed continuations were exploited to enhance the functionality of web servers and overcome the stateless nature of HTTP interaction. By comparison, the RPC chain is a generic mechanism that is independent of the service provided by servers. RPC chains support complex chaining structures, and can be used with a diverse set of servers.

The above mentioned ideas for code mobility, and others, are leveraged in a variety of high-level programming paradigms for distributed execution. Distributed workflows, e.g., [7, 75], can use distributed continuations to distribute a workflow description in a decentralized fashion.

MapReduce [23], and Dryad [76] are programming models for data-parallel jobs, such as a data mining calculations, which process large amounts of data in batches. These systems target self-contained jobs that execute for substantial periods, while RPC Chains are intended for short-lived remote executions in an environment with many diverse services that are possibly developed independently of their applications.

Mobile agents have been extensively studied in the literature and many systems have been built, including Telescript/Odyssey [72], Aglets [6], D’Agents [28], and others (see e.g., [73, 32]). A mobile agent is a process that can autonomously migrate itself from host to host as it executes; migration involves moving the process’s current state to the new host and resuming execution. The

motivation for mobile agents include (a) bringing processes closer to the resources they need in a given stage of the computation, and (b) allowing clients to disconnect from the network while an agent executes on their behalf. An RPC Chain can be considered as a mobile agent whose purpose is to execute a series of RPC calls. However, mobile agents are much more general and ambitious than RPC Chains (which possibly contributed to their eventual demise): they have social abilities, being able to adjust their behavior according to the host in which they are currently executing; they can learn about execution environments never envisioned by their creators; and they can persist if the clients that created them disappear. Much of the literature regarding mobile agents is about security (how agents can survive malicious hosts, and how hosts can protect themselves against malicious agents) and language support for code mobility (how to write programs that can transparently move to other machines). For RPC chains, security is a smaller concern in the trusted data center and enterprise environments that we consider, and we are not concerned about transparent mobility.

Some related work includes more targeted uses of mobile code. Work on Active Networks introduced network packets called *capsules*, which carry code that network switches execute to route the packet (see [70] for a survey). This provides a general scheme for extending network protocols beyond the existing deployed base, and allows for more dynamic routing schemes. In contrast, RPC chains are aimed at higher-level applications, and their main purpose is to eliminate communication hops when a client needs to call many services in succession.

Distributed Hash Tables (e.g., Chord [68], CAN [62], Pastry [63],

Tapestry [77]) have a *lookup* protocol, for finding the host responsible for a given key. Such protocols generally need to contact several hosts successively, and this can be done in two ways. In an *interactive* lookup protocol, the host that initiates the lookup operation issues RPC's to each host in succession. A *recursive* lookup protocol [26] works like a routing protocol: the host that initiates the operation contacts the first host in the sequence, which in turn contacts the next one, and so forth; when a host finds the key, it contacts the request initiator directly. This protocol is hard-coded into the lookup operation, and it is executed by a set of servers that implement this operation. In contrast, RPC chains provide a generic chaining mechanism that is independent of the operation (service function) executed.

Finally, SOAP [74] is a protocol that supports RPC's using XML over HTTP. It has the notion of intermediaries that can process a SOAP message (RPC) before it reaches the final recipient. However, there is no client logic that routes and transform messages, and the notion of a pre-specified distinguished final recipient is inherent to SOAP. Typical uses for intermediary nodes include blocking messages (firewall), buffering and batching of messages, tracing, and encrypting/decrypting messages as it passes through an untrusted domain.

CHAPTER 2

THE BUILDING BLOCKS OF CONSENSUS

Consensus protocols are a critical component in the construction of fault tolerant systems. In this chapter, we look at consensus protocols for Internet-like systems in which there are no real-time bounds on execution or message latency. Such systems are often termed *asynchronous*. While published asynchronous consensus protocols may at first appear complex and quite different from each other, we claim that all these protocols are derived from the same simple *genes*.

Our contributions are the following:

- We present the genes of consensus algorithms in the form of a skeleton algorithm that can be configured to produce various consensus protocols. The skeleton algorithm gives insight into how consensus protocols work, and we learn that consensus requires not one but two separate quorum systems;
- We demonstrate how this approach can be used to instantiate three well-known consensus protocols: Paxos [42], Chandra-Toueg [17], and Ben-Or [9];
- We implement our approach and present a performance comparison of these protocols under varying workload and crash failures. We learn interesting trade-offs between various design choices in consensus algorithms.

2.1 The Consensus Problem

In prior work, there has been neither agreement nor consensus on terminology. We propose nomenclature for talking about the consensus problem and protocols to solve it (see Table 2.1).

In the consensus problem there is a set of *proposers*, each of which can propose a *proposal*, and a set of *deciders*, each of which *decides* one of the proposals. The goal is to ensure each non-faulty decider decides the same proposal, even in the face of faulty proposers.

We must specify the execution and failure model of *processors* (computers that run programs) and *links* (network connections between processors). Processors run *actors*, which are either proposers or deciders. A processor may run both a proposer and a decider—in practice, the proposer often would like to learn the outcome of the agreement.

Processors are either *honest*, executing programs faithfully, or *Byzantine* [43], exhibiting arbitrary behavior. We will also use the terms *correct* and *faulty*, but not as alternatives to honest and Byzantine. A correct processor is an honest processor that always eventually makes progress. A faulty processor is a Byzantine processor or an honest processor that has crashed or will eventually crash. Note that honest and Byzantine are mutually exclusive, as are correct and faulty. However, a processor can be both honest and faulty.

We assume that each pair of processors is connected by a *link*, which is a bi-directional reliable virtual circuit that ensures messages sent on this link are delivered, eventually, and in the order in which they were sent (*i.e.*, an honest

sender keeps retransmitting a message until it receives an acknowledgment or crashes). Also, the receiver can tell who sent a message (*e.g.*, using MACs), so a Byzantine processor cannot forge a message so it appears like a message sent by an honest processor.

Because our system is asynchronous, we do not assume timing bounds on execution of programs or on latency of communication. We also do not assume that a processor on one side of a link can determine whether the processor on the other side of the link is correct or faulty. Timeouts cannot reliably detect faulty processors in an asynchronous system, even if only crash failures are allowed.

Why is consensus hard? Consider the following strawman protocol: each decider collects proposals from all proposers, determines the minimum proposal from among the proposals it receives (in case it received multiple proposals), and decides on that one. If there were no faulty processors at all, such a protocol would work, albeit limited in speed by the slowest processor or link.

Unfortunately, even if only crash failures are possible, deciders do not know how long to wait for proposers. If deciders were to use time-outs, they might time-out on different proposers, and these deciders would decide different proposals as a result. Thus each decider has no choice but to wait until it has received a proposal from all proposers. If one of the proposers is faulty, such a decider will never decide.

In an asynchronous system with crash failures (Byzantine failures comprise crash failures), there exists no deterministic protocol in which all correct deciders eventually decide [25] (a result called FLP after the people that showed this impossibility, Fisher, Lynch, and Patterson). We can circumvent this limi-

tation by not requiring that all correct deciders eventually decide. Instead, we will require only that the consensus protocol cannot reach a state in which some correct decider can never decide. The strawman protocol of deciding the minimum proposal can reach a state in which deciders wait indefinitely for a faulty proposer, and is, therefore, not a consensus protocol, even with our weaker requirement.

A protocol that solves the consensus problem must have the following three properties:

Definition: Agreement: If two honest deciders decide, they decide the same proposal.

Definition: Validity: If all honest proposers propose the same proposal v , then an honest decider that decides will decide v .

Definition: Non-Blocking: Given any run of the protocol that reaches a state in which a particular correct decider has not yet decided, there exists a continuation of the run in which that decider does decide on a proposal.

To abstract the interaction that takes place in agreement algorithms we describe a mechanism through which messages are exchanged. The mechanism is based on the notion of *quorum systems*, which we now review and extend.

<i>term</i>	<i>description aka</i>
actor	proposer, decider, selector, or registrar
Byzantine	potentially exhibits arbitrary behavior
correct	honest and makes progress eventually
crashed	halted execution indefinitely
decider	actor who wants to decide
faulty	Byzantine or stops making progress
guarded set	contains at least one honest participant
honest	not Byzantine (but may crash)
instance	phase in protocol
(network) link	connects two participants
processor	hardware on which actor runs
participant	member of a quorum system
proposal	initial value submitted to protocol
proposer	actor whose role is to propose a value
quorum (set)	any two quorums of participants intersect
quorum system	structure on set of actors
registrar	actor that prevents multiple decisions
selector	actor that suggests decisions
suggestion	proposal + instance identifier
max-wait set	maximal set of participants one can wait for

Table 2.1: Terminology.

2.2 Extended Quorum Systems

Before we introduce our skeleton algorithm, we introduce a useful building block. An *extended quorum system* is a quadruple $(\mathcal{P}, \mathcal{M}, \mathcal{Q}, \mathcal{G})$. \mathcal{P} is a set of processors called the *participants*. \mathcal{M}, \mathcal{Q} , and \mathcal{G} are each a collection of subsets of participants (that is, each is a subset of $2^{\mathcal{P}}$). \mathcal{M} is the collection of *maximal-wait sets*, \mathcal{Q} the collection of *quorum sets*, and \mathcal{G} the collection of *guarded sets*. Each is defined below.

A subset of \mathcal{P} is a *guarded set* if and only if it is guaranteed to contain at least one honest participant. Note that a guarded set may consist of a single

participant that may be crashed but is not Byzantine. As an example, consider an (n, t) -threshold quorum system, a system with n participants, of which at most t are faulty. In the case of crash failures only, the guarded sets are all non-empty subsets of \mathcal{P} . In the Byzantine case, guarded sets have to be of size larger than t in order to guarantee that they contain at least one honest participant.

When requesting information from all participants, crashed or Byzantine participants may never respond. An actor often tries to collect as many responses to a broadcast request as possible, but has to stop collecting additional responses when it is in danger of waiting indefinitely. \mathcal{M} characterizes this—it is a set of subsets of \mathcal{P} , none contained in another, such that some $M \in \mathcal{M}$ contains all the correct processors.¹ In a threshold quorum system, the maximal-wait sets are all subsets of $n - t$ participants, where n is the number of processors and t is the maximum number of failures.

Quorums are sets of processors that satisfy Consistency:

Definition: Consistency: An extended quorum system satisfies *Consistency* if and only if the intersection of any two quorums (including a quorum with itself) is guaranteed to contain an honest participant. (In other words, the intersection of two quorums is a guarded set.)

To illustrate, consider how consistency can be satisfied in an (n, t) -threshold system. With only crash failures possible, consistency requires that quorums be larger than $n/2$ participants, because two strict majorities always intersect, and no participants are Byzantine. When Byzantine failures are possible, con-

¹For those familiar with Byzantine Quorum Systems [47], \mathcal{M} is the set of complements of the fail-prone system \mathcal{B} . For the purposes of this chapter, it is often more convenient to talk about maximal-wait sets.

	Crash	Byzantine
guarded set (in \mathcal{G})	> 0	$> t$
quorum set (in \mathcal{Q})	$> n/2$	$> (n + t)/2$
maximal-wait set (in \mathcal{M})	$= n - t$	$= n - t$
set of participants (\mathcal{P})	$> 2t$	$> 5t$

Table 2.2: Size requirements for Threshold Quorum Systems that satisfy consistency and opaqueness.

sistency requires having quorums be larger than size $(n + t)/2$. To see why this works, consider any two quorums. Without removing duplicates, the sets together have more than $n + t$ elements. But there are only n participants, so the intersection must contain more than t participants, and thus contains at least one honest participant.

Definition: Availability: An extended quorum system satisfies *Availability* if and only if every maximal-wait set contains a quorum.

Because Availability requires that every maximal-wait set contains a quorum, we get $n - t > n/2$, or $n > 2t$, putting a lower bound on n . We obtain that $n - t > (n + t)/2$, or $n > 3t$.

Availability requires that quorums can have no more than $n - t$ elements, otherwise there would exist maximal-wait sets that do not contain a quorum.

In this chapter, we will also require that extended quorum systems satisfy *Opaqueness* [47]:²

Definition: Opaqueness: An extended quorum system satisfies *Opaqueness* if

²Opaqueness is a stronger property than *Availability* typically required of quorum systems, as Availability only requires that each maximal-wait set contain a quorum, possibly including Byzantine members.

and only if each maximal-wait set contains a quorum consisting entirely of honest participants.

We illustrate what this requirement means for (n, t) -threshold quorum systems. In a crash failure model, opaqueness requires that every maximal-wait set contains a quorum, so we get that $n - t > n/2$, or $n > 2t$, placing a lower bound on n . In the Byzantine model, t of the participants in a maximal-wait set may be Byzantine. Thus we have to require that $n - 2t > (n + t)/2$, which simplifies to $n > 5t$. Table 2.2 summarizes requirements for \mathcal{P} , \mathcal{M} , \mathcal{Q} , and \mathcal{G} in (n, t) -threshold systems.

The simplest example of quorum systems are threshold quorum systems, but other quorum systems may be more appropriate for particular applications. See [59] and [47] for advantages and disadvantages of various quorum systems for crash and arbitrary failure models respectively.

One degenerate extended quorum system, used in some well-known consensus protocols, is a *leader extended quorum system*: There is one participant (the leader), and that participant by itself forms the only maximal-wait set in \mathcal{M} , quorum in \mathcal{Q} , and guarded set in \mathcal{G} . Because quorum sets have to satisfy consistency, the leader has to be honest.

2.3 The Consensus Skeleton

In Section 2.1, we presented a strawman consensus protocol that, in the presence of faulty processors, may reach a state where deciders can never decide. To avoid this, consensus protocols invoke multiple *instances*, where an instance is

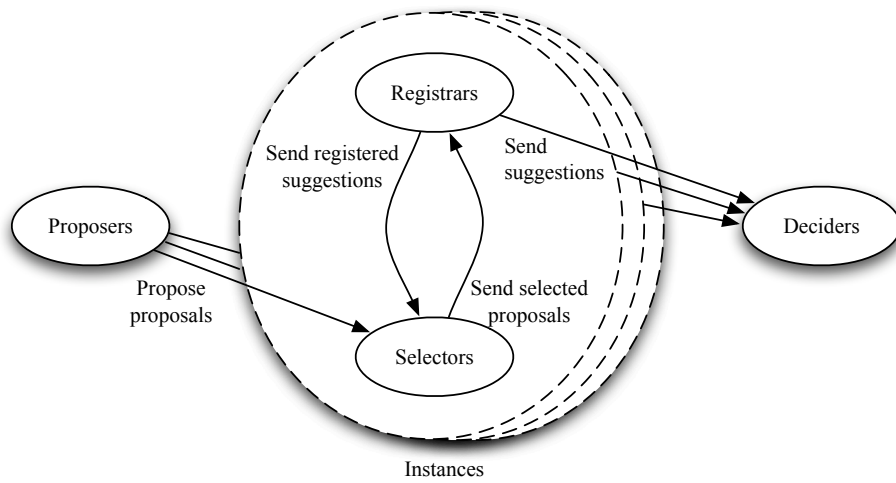


Figure 2.1: The actions of the various actors.

an execution of a protocol that, once started, runs in isolation. Instances have also been called *rounds*, *phases*, or *ballots*.

In order to guarantee consistency among decisions of deciders in the presence of multiple instances, we introduce two new types of actors in addition to proposers and deciders, namely *registrars* and *selectors*.³ A proposer sends its proposal to the selectors. Selectors and registrars exchange messages and occasionally registrars inform deciders about potential values for decision. Deciders apply some filter to reach a decision.

The actions of the various actors are summarized in Figure 2.1. Broadly speaking, the objective of selectors is to reach a decision within an instance, while the objective of registrars is to maintain a collective memory that ensures that decisions are maintained across instances, preventing conflicting decisions.

³As stated before, a processor may run multiple actors, although each can run at most one registrar and at most one selector.

2.3.1 Instances

An instance decides a proposal if an honest decider decides a proposal in that instance. All honest deciders that decide in an instance must be guaranteed to decide the same proposal, so an instance cannot decide multiple proposals. It is not guaranteed that an instance decides any proposal. By having multiple instances, if one instance does not decide, then future ones may decide. It is important to guarantee that if multiple instances decide, they decide the same proposal.

Instances are identified by instance identifiers r from a totally ordered set that we will call $\bar{\mathbb{N}}$ (can be, but does not have to be, \mathbb{N}). Instance identifiers induce an ordering on instances, and we say that one instance is *before* or *after* another instance, but keep in mind that instances may execute concurrently.

We name proposals v, w, \dots . Within an instance, proposals are paired with instance identifiers. We call a pair (r, v) a *suggestion*, where v is the proposal and r an instance identifier. A special suggestion \perp is used to indicate absence of a specific proposal.

Selectors *select* proposals, and registrars *register* suggestions. Each registrar keeps careful track of the last suggestion that it has registered. The initial registered suggestion of a registrar is \perp .

A new instance starts with the registrars sending their current registered suggestions to the selectors. (Exactly how an instance starts depends on the complete design of the consensus protocol, and will be addressed later.) Each selector determines if one of the suggestions it receives could have been decided in a previous instance. If so, it selects the corresponding proposal (of which

there can be at most one). If not, it selects one of the proposals issued by the proposers. The selector creates a suggestion from the selected proposal using the current instance identifier, and sends the suggestion to the registrars.

If a registrar receives the same suggestion from a quorum of selectors (that is, all selectors in the quorum sent the same instance identifier and proposal), it (i) registers the suggestion, and (ii) broadcasts the suggestion to the deciders. If a decider receives the same suggestion from a quorum of registrars, the decider decides the corresponding proposal in those suggestions.

2.3.2 Guarded Proposal

Selectors have to be careful not to select proposals that could conflict with prior decisions. Before selecting a proposal in an instance, a selector obtains a set of suggestions L from each participant in a maximal-wait set of registrars. A proposal v is a *potential-proposal* if L contains suggestions containing v from a guarded set. This means that at least one honest registrar sent a suggestion containing v . The selector computes the *guarded proposal* of L , if any, as follows:

1. Consider each potential-proposal v separately:
 - (a) Consider all subsets of suggestions containing v from guarded sets of registrars. The minimum instance identifier in a subset is called a *guarded-instance-identifier*;
 - (b) The maximum among the guarded-instance-identifiers for v is called the *associated-instance-identifier* of v . (Note that because v is a potential-proposal, there has to be at least one guarded-instance-identifier and

thus the maximum is well-defined.) The *support-sets* for v are those subsets of suggestions for which the guarded-instance-identifier is the associated-instance-identifier;

2. Among the potential-proposals, select all proposals with the maximal associated-instance-identifier. If there is exactly one such potential-proposal v' , and $v' \neq \perp$, then this is the guarded proposal. Otherwise there is no guarded proposal.

For example, consider a Byzantine threshold quorum system Q with $t = 2$ and $n = 11$. Thus, a quorum has to have more than $(n + t)/2$ elements, which is 7 or more in our case, and a maximal-wait set has to contain at least $n - t = 9$ elements. A guarded set has at least $t + 1 = 3$ elements. Now consider the following suggestions in the maximal-wait set: 4 suggestions with the value $(3, \text{green})$, 2 suggestions with the value $(5, \text{green})$, and 3 suggestions with the value $(4, \text{red})$. Both `green` and `red` are potential-proposals. The maximum instance identifier among sets of 3 suggestions for `green` is 3, as there are only 2 suggestions with instance identifier 5. The maximum instance identifier among sets of 3 suggestions for `red` (there is only one such set) is 4, thus 4 is the maximum associated-instance-identifier. Because `red` is the only proposal with an associated-instance-identifier of 4, `red` is the guarded proposal even though there are twice as many suggestions supporting `green`!

For benign systems, the guarded proposal in an (n, t) -threshold quorum system is simply the proposal corresponding to the maximum instance identifier in L . For Byzantine systems, divide the suggestions in L by proposal, and sort each subset by instance identifier. Consider the proposal with the highest $t + 1^{\text{st}}$ instance identifier. If there is exactly one of these, this is the guarded proposal.

Note that in protocols in which instances are invoked one after the other completes, sequentially, *associated-instance-identifier* will always be the identifier of the previous instance.

If a decider obtains suggestions (r, v) from a quorum of registrars (and consequently decides), then any honest selectors in instances $\geq r$ are guaranteed to compute a guarded proposal v' such that $v' = v$ (unless they crash). If a selector fails to compute a guarded proposal in a particular instance then this is both evidence that no prior instance can have decided and a guarantee that no prior instance will ever decide. However, the reverse is not true. If a selector computes a guarded proposal v' , it is not guaranteed that v' is or will be decided.

2.3.3 Extended Quorum System API

Participants of an extended quorum system send and receive messages of the form $\langle \text{message-type}, \text{instance}, \text{source}, \text{suggestion} \rangle$. The source indicates the sending processor.

An extended quorum system $\mathcal{E} = (\mathcal{P}, \mathcal{M}, \mathcal{Q}, \mathcal{G})$ has the following interface:

- $\mathcal{E}.\text{broadcast}(m)$: send a message m to all participants in \mathcal{P} ;
- $\mathcal{E}.\text{wait}(\text{pattern})$: wait for messages, matching the given pattern (specifies, for example, the message type and instance number). When the sources of the collected messages form an element or a superset of an element of \mathcal{M} , then return the set of collected messages;
- $\mathcal{E}.\text{uni-quorum}(\text{set of messages})$: if the set of messages contains the same

- suggestion from a quorum, then return that suggestion.⁴ Otherwise, return \perp ;
- $\mathcal{E}.\textit{guarded-proposal}(\text{set of messages})$: return the guarded proposal among these messages, or \perp if there is none.

2.3.4 State Maintenance

Each selector and registrar is associated with an instance, which is the instance it is currently sending messages in, and receiving messages from. A registrar can change the instance with which it is associated, progressing to a later instance after aborting the current one. A selector can also progress to later instances, but unlike a registrar it is allowed to keep participating in older instances.

The instance protocol uses *not one but two extended quorum systems*:

1. Registrars form an extended quorum system \mathcal{R} that is the same for all instances. \mathcal{R} has to satisfy consistency and opaqueness. Selectors use \mathcal{R} to find the guarded proposal, if any, to select proposals that do not conflict with earlier decisions.
2. Selectors form an extended quorum system \mathcal{S}^r , which may be different for each instance r . Each \mathcal{S}^r has to satisfy both consistency and opaqueness as well. Registrars in instance r use quorums of \mathcal{S}^r to avoid having two registrars register different suggestions within the same instance.

Deciders, although technically not part of an instance, do try to obtain the same suggestion from a quorum of registrars in each instance. We associate de-

⁴By the consistency property, there can be at most one such suggestion.

cidars with instances for simplicity of presentation. Also for convenience, we will have deciders form an extended quorum system \mathcal{D} . However, deciders never send messages, and thus we will only use \mathcal{D} to send messages to all deciders.

Each selector i maintains a set P_i containing proposals it received (across instances). A selector waits for at least one proposal before participating in the rest of the protocol, so P_i is never empty during execution of the protocol. (Typically, P_i first contains a proposal from the proposer on the same processor as selector i .) For simplicity we assume an honest proposer sends a single proposal. The details of how P_i is formed and used are different for different agreement protocols, so this will be discussed when the full protocols are presented. P_i has an operation $P_i.pick(r)$ that returns either a single proposal from the set or some value as a function of r . Different protocols use different approaches for selecting that value, and these will also be discussed later.

Registrars are assumed to have state that survives crashes and recoveries. In particular, a registrar j running on an honest processor maintains:

r_j : current instance identifier;

c_j : last registered suggestion, initially \perp .

Both increase monotonically over time.

2.3.5 The Instance Mechanism

A protocol execution is a collection of instance executions. Figure 2.2 shows the mechanism that enables the execution of an individual instance. We call this the *instance mechanism*. A protocol execution starts when the first instance execution starts. How instances are invoked is different for different protocols, and some ways are described in Section 2.4. The extended quorum systems used in the instance protocol determines the potential participants in each instance and is also protocol specific. Potential participants execute in an instance once pre-conditions for their actions are satisfied.

Lemma 1. *In the mechanism in Figure 2.2,*

- (a) *if any honest registrar i computes a suggestion $q_i^r \neq \perp$ in Step (R.2) of instance r , then any honest registrar that computes a non- \perp suggestion in that step of that instance, computes the same suggestion.*
- (b) *if any honest decider k computes a suggestion $d_k^r \neq \perp$ in Step (D.2) of instance r , then any honest decider that computes a non- \perp suggestion in that step of that instance, computes the same suggestion.*

Proof. Case (a): in the set of suggestions contained in M_i^r collected in Step (R.1) \mathcal{S}^r .*uni-quorum* must have identified a unanimous suggestion $q_i^r \neq \perp$ from a quorum of selectors Q_i^r . Consider another honest registrar j that completes Step (R.2) of instance r , computing a unanimous suggestion q_j^r for $q_j^r \neq \perp$ from a quorum Q_j^r . By the consistency property on \mathcal{S}^r , $Q_i^r \cap Q_j^r$ contains an honest selector. An honest selector always sends the same suggestion to all registrars, implying $q_i^r = q_j^r$.

At the **start of instance** r , each **registrar** i executes:

(R.0) send c_i to all participants (selectors) in \mathcal{S}^r :

$$\mathcal{S}^r.\text{broadcast}(\langle \text{select}, r, i, c_i \rangle)$$

Each **selector** j in \mathcal{S}^r executes:

(S.1) wait for `select` messages from registrars:

$$L_j^r := \mathcal{R}.\text{wait}(\langle \text{select}, r, *, * \rangle);$$

(S.2) see if there is a guarded proposal:

$$v_j^r := \mathcal{R}.\text{guarded-proposal}(L_j^r);$$

(S.3) if not, select from received proposals instead:

$$\text{if } v_j^r = \perp \text{ then } v_j^r := P_j.\text{pick}(r) \text{ fi};$$

(S.4) send a suggestion to all registrars:

$$\mathcal{R}.\text{broadcast}(\langle \text{register}, r, j, (r, v_j^r) \rangle);$$

Each **registrar** i (still in instance r) executes:

(R.1) wait for `register` messages from selectors:

$$M_i^r := \mathcal{S}^r.\text{wait}(\langle \text{register}, r, *, * \rangle);$$

(R.2) see if there is a unanimous suggestion from a quorum:

$$q_i^r := \mathcal{S}^r.\text{uni-quorum}(M_i^r);$$

(R.3) `register` the suggestion:

$$c_i := \text{if } q_i^r = \perp \text{ then } (r, \perp) \text{ else } q_i^r \text{ fi};$$

(R.4) send the suggestion to all deciders:

$$\mathcal{D}.\text{broadcast}(\langle \text{decide}, r, i, c_i \rangle)$$

Each **decider** k executes:

(D.1) wait for `decide` messages from registrars:

$$N_k^r := \mathcal{R}.\text{wait}(\langle \text{decide}, r, *, * \rangle);$$

(D.2) see if there is a unanimous suggestion from a quorum:

$$d_k^r := \mathcal{R}.\text{uni-quorum}(N_k^r);$$

(D.3) if there is, and not \perp , decide:

$$\text{if } (d_k^r = (r, v')) \text{ and } v' \neq \perp \text{ then decide } v' \text{ fi};$$

Figure 2.2: An instance of the consensus protocol.

The case for the deciders (b) is similar although it is based on the consistency property on \mathcal{R} , since the deciders require a unanimous suggestion from a quorum of registrars in instance r before deciding. \square

Note that Step (S.2) does *not* have the property of Lemma 1 because selectors do not try to obtain a unanimous suggestion from a quorum.

Corollary 2. *In the mechanism in Figure 2.2, if any honest registrar registers a suggestion (r, v) with $v \neq \perp$ in Step (R.3) of instance r , then any honest registrar that registers a suggestion with non- \perp proposal in that step of that instance, registers the same suggestion.*

Proof. By Lemma 1 all honest registrars, unless they crash, compute the same suggestion or \perp in (R.2). Those that computed a non- \perp suggestion therefore register the same suggestion in (R.3), and send that same non- \perp suggestion in (R.4) of that instance. \square

Lemma 3. *In the mechanism in Figure 2.2, if any honest registrar sends a suggestion (\bar{r}, v) with $v \neq \perp$ in Step (R.0) of instance r (r is not the initial instance), then any honest registrar that sends a suggestion (\bar{r}, v') with $v' \neq \perp$ in that step of that instance, sends the same proposal, i.e., $v = v'$.*

Proof. By Corollary 2 all honest registrars that register a suggestion in instance \bar{r} based on the same non- \perp proposal register the same suggestion in (R.3). Therefore, if their suggestions sent in (R.0) of instance r were registered with a non- \perp suggestion in (R.3) of the same instance \bar{r} , they send the same suggestions. \square

Lemma 4. *In the mechanism in Figure 2.2,*

- (a) *if each honest selector that completes Step (S.4) of instance r sends the same suggestion, then any honest registrar that completes Step (R.2) of that instance computes the same suggestion;*
- (b) *if each honest registrar that completes Step (R.4) of instance r sends the same suggestion, then any honest decider that completes Step (D.2) of that instance computes the same suggestion;*
- (c) *if each honest registrar that completes Step (R.0) of instance r sends the same suggestion, then any honest selector that completes Step (S.2) of that instance computes the same proposal.*

Proof. Case (a): assume the conditions of the lemma with respect to each honest selector completing Step (S.4) of instance r . Each registrar that completes Step (R.1) collects suggestions from a maximal-wait set of selectors. By definition, each quorum contains at least one honest selector. Therefore, the set M_i^r of instance r of any honest registrar i cannot contain a unanimous suggestion for a suggestion different from what all the honest selectors have broadcast in Step (S.4). Opaqueness requires that every maximal-wait set contains a quorum of honest participants, therefore, every honest registrar that completes Step (R.2) of instance r will end up having a unanimous suggestion from a quorum of selectors, and will compute the same suggestion.

Case (b) is similar to the first case, and follows because each honest decider will eventually receive `decide` messages for the same suggestion sent by a quorum of honest registrars in Step (R.4).

We now prove case (c) in which each honest registrar that completes Step (R.0) sends the same registered suggestion. Notice that each set of sug-

gestions L_i^r contains at least the suggestions from a guarded set of honest registrars, and cannot contain suggestions from a guarded set of Byzantine registrars. Since each honest registrar sends the same suggestion, no honest selector will find a unanimous suggestion from a guarded set of registrars for any proposal contained in the suggestion of the honest registrars. Each selector will have a unanimous suggestion sent by a guarded set of honest registrars, and therefore will end up computing the same proposal at the end of Step (S.2). \square

And now we address the last and most important property we need to prove:

Lemma 5. *The mechanism in Figure 2.2 satisfies that if r' is the earliest instance in which a proposal w is decided by some honest decider, then for any instance r , $r > r'$, if an honest registrar registers a suggestion in Step (R.3), it is (r, w) .*

Proof. Since all instances are taken from a fully ordered set, any subset of them are fully ordered. The proof will be by induction on all the instances, past instance r' , in which eventually some honest registrar registers a suggestion.

Let $w \neq \perp$ be the proposal decided by an honest decider in Step (D.2) of instance r' . Let $Q' \in \mathcal{R}$ be the quorum in instance r' whose suggestions caused the decider to decide w .

Let $r_1 > r'$ be the first instance past r' at which some honest registrar eventually completes Step (R.3). Since this registrar completes Step (R.3), it must have received `register` messages from a maximal-wait set of selectors following Step (R.1) of instance r_1 . Each honest selector that sent such a message received `select` messages from a maximal-wait set of registrars that were sent in their Step (R.0) of instance r_1 . Each honest registrar that completes Step (R.0) did not register any new suggestion in any instance $r'', r' < r'' < r_1$, because r_1 is the first

such instance. Moreover, the registrar will not register such a suggestion in the future, since it aborted all such instances r'' before sending its `select` message in Step (R.0) of instance r_1 .

In Step (R.0) a registrar sends the last suggestion it had registered. Some registrars may send suggestions they had registered prior to instance r' while some other registrars send suggestions they registered in Step (R.4) of instance r' .

Each honest selector j awaits a set of messages L_j from a maximal-wait set in Step (S.1). L_j has to contain suggestions from a quorum Q^{r_1} consisting entirely of honest registrars (by the opaqueness property of \mathcal{R}). By the consistency property of \mathcal{R} the intersection of Q^{r_1} and $Q^{r'}$ has to contain a guarded set, and thus Q^{r_1} has to contain suggestions from a guarded set of honest registrars that registered (r', w) . There cannot be such a set of suggestions for a later instance, prior to r_1 . By Corollary 2 and Lemma 3, there cannot be any suggestions from a guarded set for a different proposal in instance r' . Thus, each honest selector will select a non- \perp proposal and those proposals are identical.

By Lemma 4, every honest registrar that completes Step (R.4) will register the same suggestion, thus the proof holds for r_1 .

Now assume that the claim holds for all instances $r'', r' < r'' < r$, and we will prove it for instance r .

There is an honest registrar that completes Step (R.3) in instance r and registers (r, w) . Following Step (R.1) of instance r it must have received `register` messages from a maximal-wait set of selectors. Each honest selector that sent such a message received `select` messages from a maximal-wait set of regis-

trars that were sent in Step (R.0) of instance r .

Each honest registrar sends the last suggestion it had registered. Some honest registrars may send suggestions they had registered prior to instance r' while some other honest registrars send suggestions they registered in Step (R.4) of instance r'' , $r' \leq r'' < r$. By the induction hypothesis, all honest registrars that send a suggestion that was registered past instance r' use the proposal w in their suggestion.

In instance r , each honest selector j awaits a set of messages L_j from a maximal-wait set in Step (S.1). L_j has to contain suggestions from a quorum Q' consisting entirely of honest registrars (by the opaqueness property of \mathcal{R}). By the consistency property of \mathcal{R} the intersection of Q' and Q^r has to contain a guarded set, and thus Q' has to contain suggestions from a guarded set of honest registrars that registered (r', w) in instance r' , and may have registered (r'', w) in some later instance. Therefore, selector j obtains w as a possible potential-proposal. Since all honest registrars that register a suggestion past instance r' register the same proposal, there is a support-set for w with associated-instance-identifier $\bar{r} \geq r'$.

There cannot be any other possible potential-proposal with an associated-instance-identifier later than r' , since, by induction, no honest registrar registers a suggestion with a different proposal later than r' . Therefore, each honest selector will select the proposal w . By Lemma 4, every honest registrar that will complete Step (R.4) will register the same suggestion, thus the proof holds for r .

□

We now formulate and prove the main theorem about instances:

Theorem 6 (Agreement). *If two honest deciders decide, they decide the same proposal.*

Proof. If the deciders decide in the same instance, the result follows from Lemma 1. Say one decider decides v' in instance r' , and another decider decides v in instance r , with $r' < r$. By Lemma 5, all honest registrars that register in instance r register (r, v') . By the consistency property of \mathcal{R} , an honest decider can only decide (r, v') in instance r , and thus $v = v'$. \square

2.4 Full Protocols

The description of instances above does not specify how instances are created, how broadcasts are done in steps (R.0), (S.4), and (R.4), what specific extended quorum systems to use for \mathcal{R} and \mathcal{S}' , how a selector j obtains proposals for P_j , or how j selects a proposal from P_j . We now show how Paxos [42], the algorithm by Chandra and Toueg [17], and the early protocol by Michael Ben-Or [9] make these choices. They use the instance protocol as a subroutine.

2.4.1 Paxos

Paxos [42] was originally designed only for honest systems. In Paxos, any processor can create an instance r at any time, and it becomes the *leader* of that instance. The choice of leader is governed by a weak leader election protocol (see [42]) that is outside the scope of this chapter. The leader creates a unique instance identifier r from its processor identifier and a sequence number per processor that is incremented for each new instance created on that processor.

The leader runs both a proposer and a selector. \mathcal{S}' is a leader extended quorum system consisting only of the selector at the leader.

The leader starts the instance by broadcasting a `prepare` message containing the instance identifier to all registrars. (This broadcast is not part of the instance protocol of Figure 2.2.) Upon receipt, a registrar i first checks that $r > r_i$, and, if so, sets r_i to r and proceeds with Step (R.0). Note that since there is only one participant in \mathcal{S}_r , the broadcast in (R.0) is actually a point-to-point message back to the leader, now acting as selector. In Step (S.3), if the leader has to pick a proposal from P_j , it selects the proposal by the local proposer, thus there is no need for proposers to send their proposals to all selectors.

Validity follows directly from the absence of Byzantine participants. To see why Paxos is **Non-Blocking**, consider a state in which some correct decider has not yet decided. Now consider the following continuation of the run: one of the correct processors creates a new instance with an instance identifier higher than used before. Because there are always correct processors and there is an infinite number of instance identifiers, this is always possible. The processor sends a `prepare` message to all registrars. All honest registrars start in Step (R.0) of the instance on receipt, so the selector at the leader will receive sufficient `select` messages in Step (S.1) to continue. Due to Lemma 4, and the fact that there is only one selector in \mathcal{S}' , all honest registrars register the same suggestion in Step (R.3). The deciders will each receive a unanimous suggestion from a quorum of registrars in Step (D.1) and decide in Step (D.3).

2.4.2 Chandra-Toueg

The Chandra-Toueg algorithm is another consensus protocol that is designed for honest systems [17]. The Chandra-Toueg algorithm requires a coordinator in each instance. The role of the coordinator is similar to the leader in Paxos. However, unlike Paxos, Chandra-Toueg does not use a leader election protocol. Instead, the coordinator of each instance is defined by a simple mod of the instance number by the number of processors in the system, *i.e.*, the role of the coordinator rotates from processor to processor at the end of each instance. Each processor in the system is both a proposer and a registrar. For each instance r , the selector quorum \mathcal{S}^r is the extended quorum consisting only of the coordinator of that instance.

To start the protocol, a proposer sends a proposal message to all processors. Upon receiving the first proposal, a registrar starts in instance 0 and executes (R.0). The coordinator of each instance starts (S.0) upon receiving a `select` message for that instance. In (S.3), $P_i.pick(r)$ will simply return the first proposal that was received by the coordinator. Registrars that successfully complete (R.1-4) move to the next instance.

Note that when registrars are waiting for a `register` message from the selector of a particular instance, they are not guaranteed to receive a reply because the coordinator of that instance can fail. Registrars thus have to be prepared to timeout. When this happens, the registrars start executing (R.0) in the next instance, which would have a different coordinator. When a registrar receives a `register` message with a larger instance number than the one it is currently in, it aborts the current instance and skips forward to the instance of the `register` message.

In the original description of the Chandra-Toueg algorithm, the coordinator for an instance is also the decider for that instance. This means that in order for all processors to become aware of a decision, the coordinator has to broadcast an announcement. We can modify the Chandra-Toueg algorithm such that all processors are deciders in all instances without affecting the rest of the protocol. The effect of this change is the elimination of one round of communication while increasing the number of messages that are sent in (R.4) of the instance mechanism. This is similar to the algorithm proposed in [57]. A comparison of the original Chandra-Toueg algorithm and this modified version was done in [71].

As in the case of Paxos, **Validity** follows directly from the absence of Byzantine participants. The **Non-blocking** property follows from that fact that a honest, correct selector can always receive sufficient `select` messages in (S.1) to continue. All honest registrars will always receive the same suggestion in (R.3) since there is only one selector in each instance. If the coordinator for an instance fails, registrars for that instance will timeout and move to the next instance.

2.4.3 Ben-Or

In this early protocol [9], each processor runs a proposer, a selector, a registrar, and a decider. Instances are numbered $1, 2, \dots$. Proposals are either 0 or 1 (that is, this is a binary consensus protocol), and each $P_i = \{0, 1\}$. $P_i.pick(r)$ selects the local proposer's proposal for the first instance, or a random one in later instances.

Each of the selectors, registrars, and deciders starts in instance 1 and runs a

loop. A selector j runs a loop consisting of steps (S.1) through (S.4), incrementing r_j right after Step (S.4). A registrar i runs a loop consisting of steps (R.0) through (R.4), incrementing r_i after Step (R.4). Note that the broadcasts in steps (R.0) and (R.4) are to the same destination processors and happen in consecutive steps, so they can be merged into a single broadcast, resulting in just two broadcasts per instance. Finally, a decider k runs a loop consisting of steps (D.1) through (D.3), incrementing r_k after Step (D.3).

\mathcal{S}' is the same as \mathcal{R} for every instance r ; both consist of all processors and uses a threshold quorum system. Ben-Or works equally well in honest and Byzantine environments as long as opaqueness is satisfied. It can be easily shown that if a decider decides, all other deciders decide either in the same or the next instance. This suggests an easy termination rule.

Validity follows from the rule that selectors select the locally proposed proposal in the first instance: If all selectors select the same proposal v , by Lemma 4 the registrars register v , and, by the opaqueness property of \mathcal{R} , the deciders decide v . The **Non-Blocking** property follows from the rule that honest selectors pick their proposals at random in all but the first instance, and so it is always possible that they pick the same proposal, after which decision in Step (D.3) is guaranteed because of the opaqueness property of \mathcal{R} .

2.5 Implementation and Protocol Comparisons

The description of the Paxos, Chandra-Toueg, and Ben-Or protocols in the previous section show that while these protocols were originally presented as different from one another, they share a common skeleton. Using the instance

mechanism presented in Section 2.3.5, each of these protocols can be instantiated by using protocol-specific ways of i) defining the selector quorums in each instance, ii) starting instances, and iii) implementing $P_i.pick(r)$ in (S.3).

Having observed the similarities between the three protocols, we now investigate the effect of their differences on their performance. To do this, we implemented the instance mechanism defined earlier, and built each of the three protocols on top of it.

In this section, we present the implementation of these protocols and results from our simulations.

2.5.1 Implementation

We built a replicated state machine out of the consensus protocols that provides a simple logging service to remote clients. The service consists of a set of servers that run a consensus protocol. Clients can submit values to any server, which will then attempt to get that value decided in an *epoch*. To decide a value, a server submits that value as a *proposal* associated with the current epoch. When a value is decided in an *epoch*, the client that submitted the value is informed of the epoch number that the value was decided in, and all servers move to the next epoch. Each server maintains an internal queue of values that it has received from clients and attempts to get them decided in a FIFO fashion.

For the implementation of Paxos, there is an important design decision that was not described in the original protocol [42]. Paxos requires a leader election mechanism that is integral to the performance of the protocol. In our imple-

mentation, we support two different leader election mechanisms. First, we built a version of Paxos where each processor that wants to propose a value simply makes itself the leader. By having each processor pick instance numbers for instances where it is the leader from a disjoint set of instance numbers, we ensure that each instance can only have one unique leader. We call this version of Paxos *GreedyPaxos*.

We also built a second variant of Paxos which uses a token passing mechanism to determine the leader. We call this version of Paxos *TokenPaxos*. Each processor i maintains state of who the current leader is in a local variable L_i . Initially, L_i is set to *null*. Upon receiving a valid `prepare` message from another processor, *i.e.*, a `prepare` message for the current epoch and with an instance number larger than any that has previously been received, the processor sets the sender of the `prepare` message as the leader. To propose a value, a processor sends the proposal to the current leader. If the current leader is not known, it makes itself the leader and starts an instance. Each `select` message is tagged by the sending registrar with a token request if that processor has pending requests in its queue and would like to receive the token in a subsequent epoch. Each processor that sees a register message updates a local bitmap to keep track of the list of processors which are requesting the token. When the current leader commits all its local requests, it sends the token to a random processor that is requesting the token. In order to recover from lost tokens from a crashed leader, each processor sets the value of L_i to *null* if it does not receive any messages from the current leader for a certain amount of time.

For the implementation of Chandra-Toueg, we modify the original algorithm such that all processors are deciders in all instances. As described in Sec-

tion 2.4.2, this avoids requiring deciders to broadcast a decision when a value is decided. This improves the performance of our particular application where all servers need to learn about decisions.

All our implementations used a simple threshold quorum system for selector, registrar, and decider quorums.

2.5.2 Experimental Setup

In all our experiments, the logging service consists of a set of 10 servers. The workload is sent to the servers via a set of 10 clients. Each client sends requests to the servers at a rate that is described by a poisson distribution with a mean of λ_c requests per minute. Client's choice of server to send a request to is decided randomly. All messages between client to server and server to server have a latency that is given by a lognormal distribution with a mean of 100 ms and a standard deviation of 20 ms. For each set of experiments, we measure the duration between the time that a server first receives a value from a client to the time that the server learns that the value has been decided.

2.5.3 Results

In the first set of experiments, we ran the server against clients with varying loads until 100 values have been decided by the logging service. We vary the request rate from each client, λ_c , from 0.5 requests per minute to 14 requests per minute. We report on the mean and median values of 100 decisions averaged over 8 runs of each experiment. (Presenting full distributions or even just error

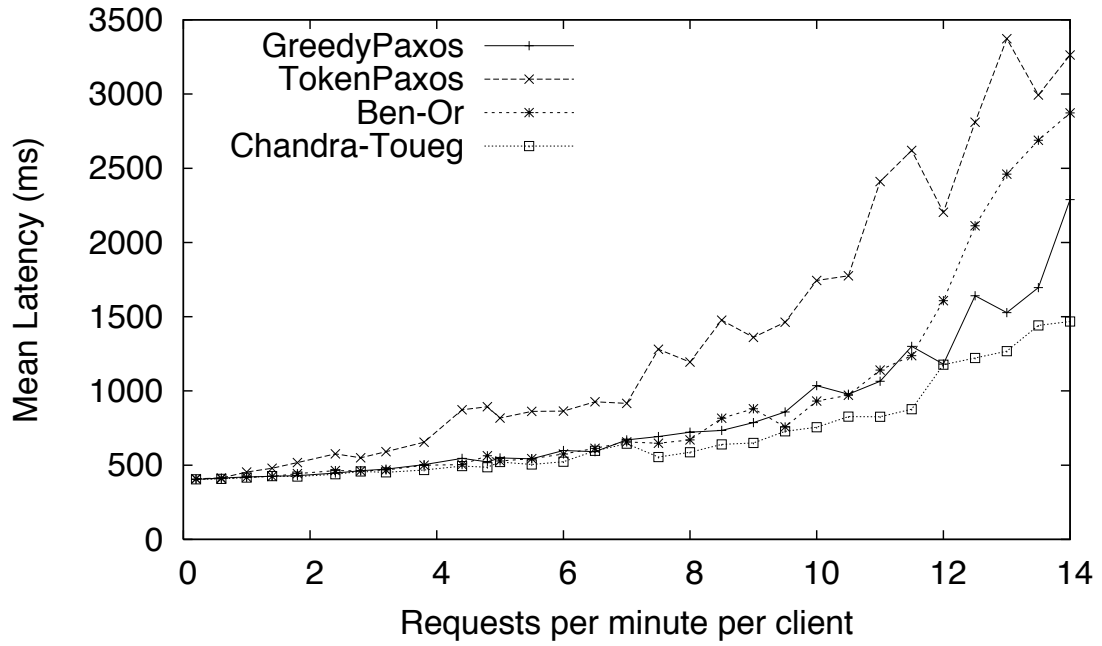


Figure 2.3: Mean time to decide a single value under varying request rates

bars makes the data difficult to read.)

Figure 2.3 and Figure 2.4 show the mean and the median time it takes for a single value to be decided. The graphs show that as load increases, the time it takes for a value to be decided increases gradually. At low loads, the performance of all four algorithms is about equivalent. This is because in all four algorithms, in the ideal case, it takes four rounds of communication for a value to be decided. This means that in the best case, it takes on average 400 ms for value to be decided.

As load is increased, performance degrades because there is contention between different servers to get different values committed in the same epochs.

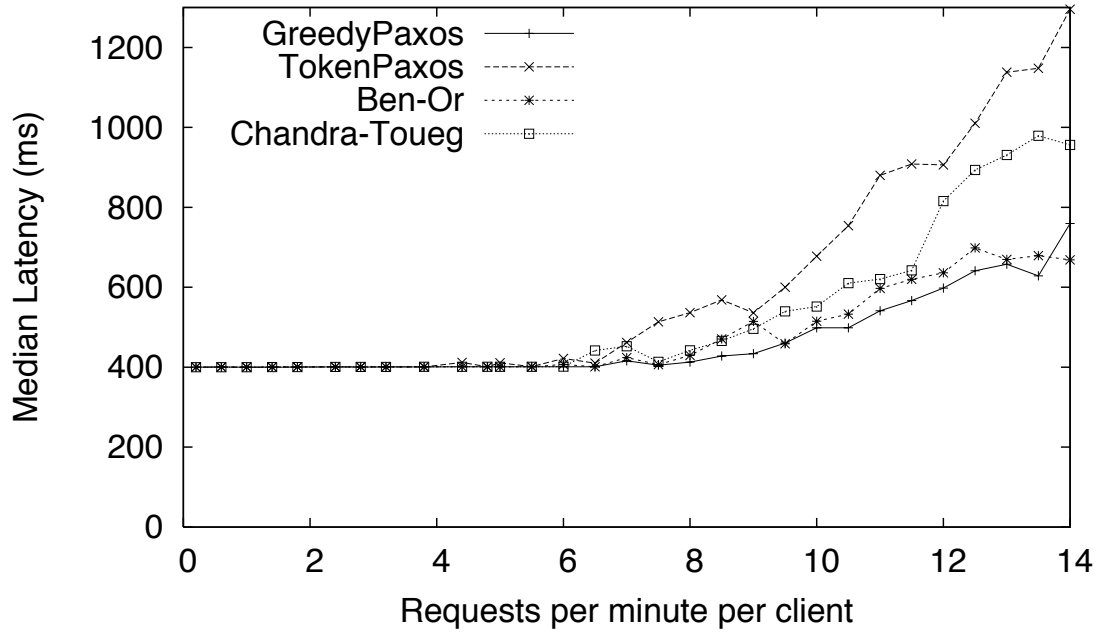


Figure 2.4: Median time to decide a single value under varying request rates

Note that GreedyPaxos performs consistently better than TokenPaxos, particularly under heavy load. This is because GreedyPaxos does not need to wait for the token before proposing a value. Under heavy load, each GreedyPaxos processor sends a `prepare` message in the beginning of each epoch without having to wait. The processor with the largest instance number wins and gets its value decided. TokenPaxos, on the other hand, will always decide values of the processor with the token before passing the token to the next processor with requests. This has 2 implications: i) if the leader keeps getting new requests, other processors can starve, and ii) one round of communication is wasted in passing the token. This results in worse performance.

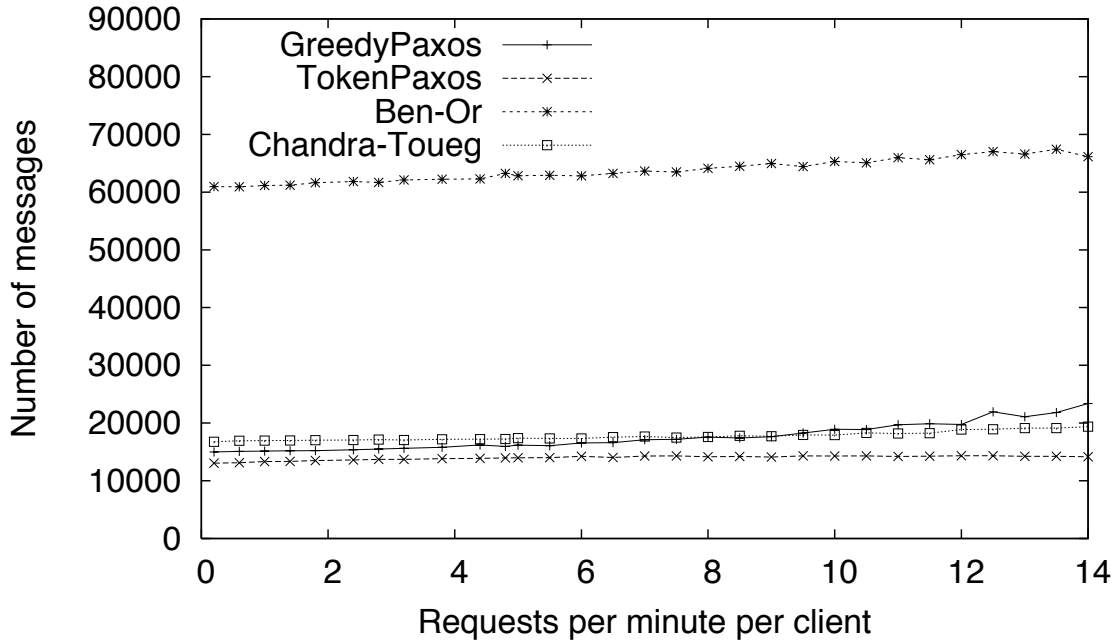


Figure 2.5: Communication overhead under varying request rates

Figure 2.5 shows the number of messages that each protocol uses to commit 100 values under different request rates. We note that Ben-Or incurs a much larger overhead than the other protocols. This is because Ben-Or uses a selector quorum that consists of all processors rather than just a leader/coordinator. This means that (R.0) and (S.4) of the instance mechanisms send n^2 messages in each instance, rather than just n messages in Paxos and Chandra-Toueg.

We also observe that compared to TokenPaxos, GreedyPaxos sends more messages as load increases. Under heavy load, each GreedyPaxos processor will broadcast a `prepare` message to all other processors in the beginning of every round. This results in n^2 messages being sent rather than the n `prepare` messages that are sent in the case of TokenPaxos. This effect is shown in Figure 2.5.

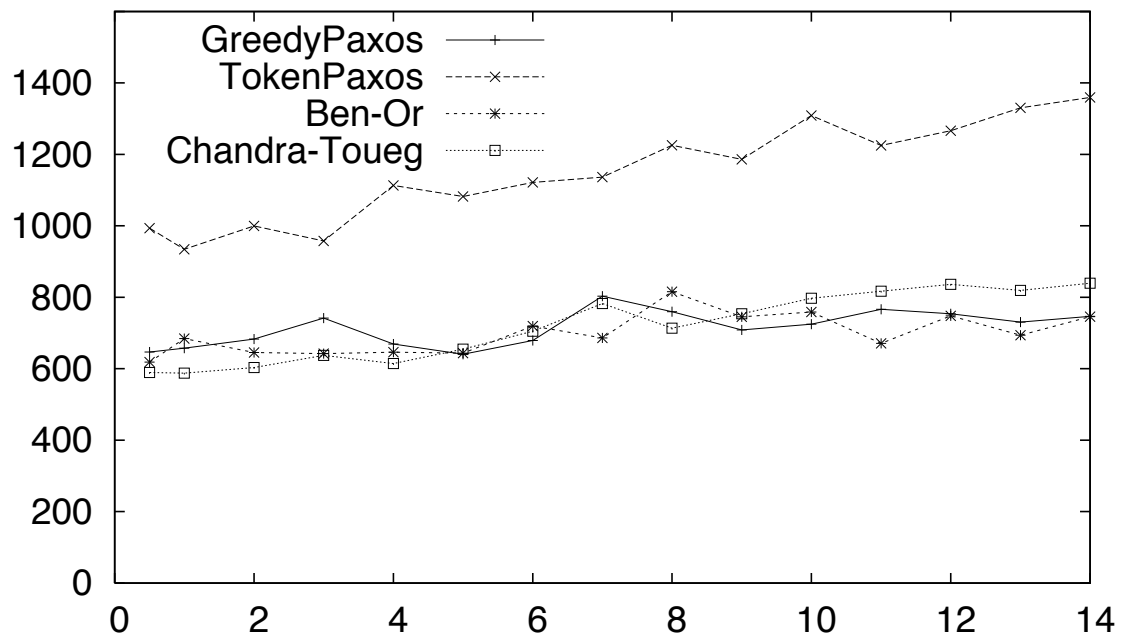


Figure 2.6: Mean time to decide a single value under varying failure rates

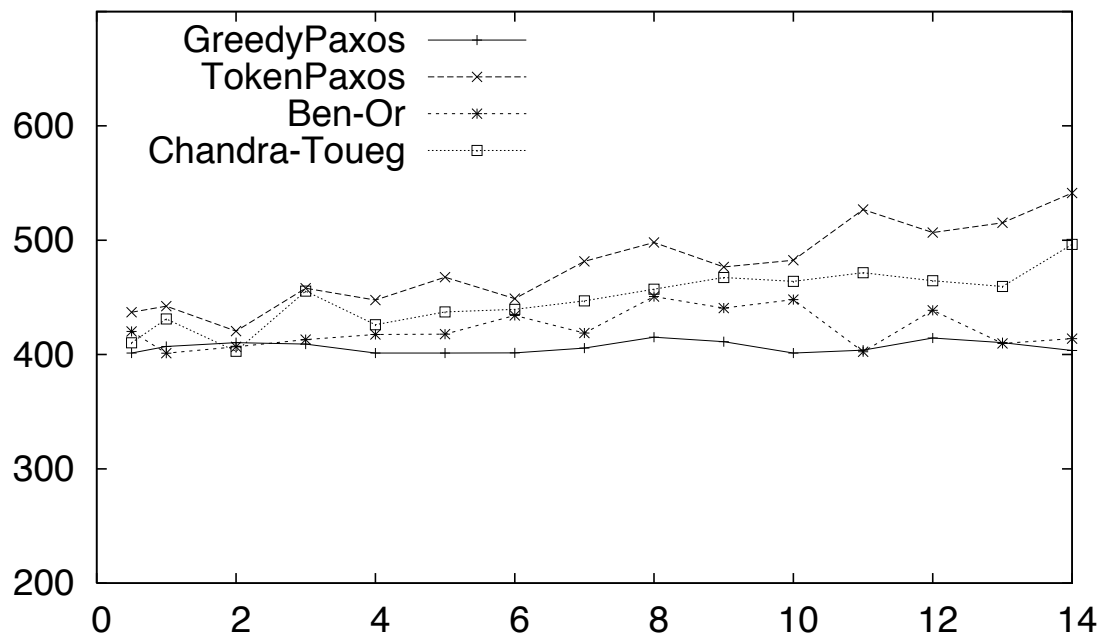


Figure 2.7: Median time to decide a single value under varying failure rates

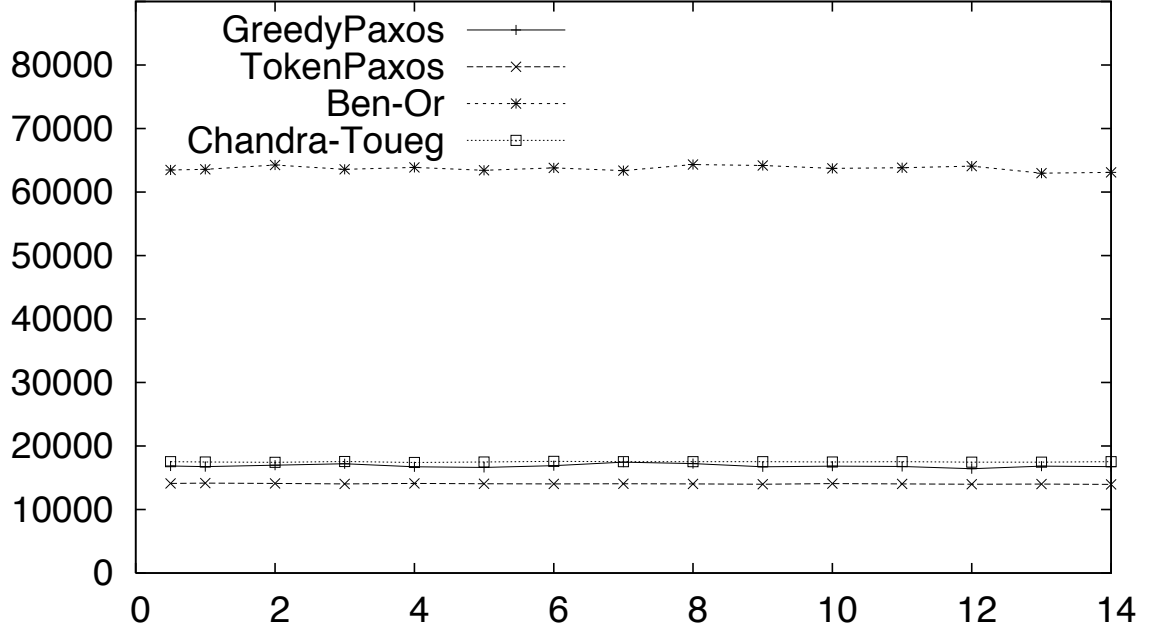


Figure 2.8: Communication overhead under varying failure rates

In order to investigate the performance of each protocol under crash failure, we injected failures into our simulations. We modeled the arrival of failure events as a poisson distribution with a rate of λ_f failures per minute. When a failure event occurs, we fail a random server until the end of the epoch. To ensure that the system is able to make progress, we limit the number of failures in an epoch to be less than half the number of servers in the system. Keeping the request rate from clients steady at 7 requests per minute per client, we vary the failure rate from 0.5 failures per minute to 12 failures per minute.

Figure 2.6 and Figure 2.7 show mean and the median decision latency, respectively, for the the four protocols under varying failure rates. Note that GreedyPaxos and Ben-Or are not affected significantly by server failures. Chandra-Toueg and TokenPaxos, on the other hand, see significant performance degradation as the failure rate increases. This is because Chandra-Toueg and To-

kenPaxos are both dependent on timeout to recover from failures of particular processors. In the case of Chandra-Toueg, the failure of the coordinator requires that all registrars timeout and move to the next instance. In the case of TokenPaxos, if the processor that is holding the token crashes, a timeout is required to generate a new token.

A comparison study presented by Hayabashibara et al. found that Paxos outperforms Chandra-Toueg [33] under crash failures. We find that this result depends on the leader election protocol used by Paxos. In our experiments, GreedyPaxos outperforms Chandra-Toueg, but TokenPaxos performs worse under certain failure scenarios.

Figure 2.8 shows the message overhead of each protocol under varying failure rate, clearly showing that the number of messages sent is not affected by failures.

CHAPTER 3

BOSCO: ONE-STEP BYZANTINE ASYNCHRONOUS CONSENSUS

In the previous chapter, we presented a skeleton algorithm that demonstrated the commonalities between several well-known consensus algorithms. In this chapter, we examine consensus algorithms for a particular setting, Byzantine asynchronous systems, where faulty processors can behave in an arbitrary manner and there are no assumptions about the relative speed of processors nor about the timely delivery of messages. We present novel algorithms that can deliver *one-step* performance under these assumptions.

Previous results have shown that algorithms that solve asynchronous Byzantine consensus must have correct executions that require at least two communication steps even in the absence of faults [39], where a single communication step is defined as a period of time where each processor can i) send messages; ii) receive messages; and iii) do local computations, in that order. However, this does not mean that such algorithms must *always* take two or more communication steps. We show that when there is no contention, it is possible for processors to decide a value in one communication step.

One-step decisions can improve performance for applications where contention is rare. Consider a replicated state machine: if a client broadcasts its operation to all machines, and there is no contention with other clients, then all correct machines propose the same operation and can respond to the client immediately. Thus an operation completes in just two message latencies, the same as for a Remote Procedure Call to an unreplicated service.

Previously, such *one-step* algorithms have been proposed for crash failure as-

sumptions [13, 11, 19, 44, 45, 24]. In this chapter, we will show that they are also possible for Byzantine asynchronous consensus in the presence of a strong network adversary. Overall, we present the following contributions in this chapter:

- we provide two definitions of one-step asynchronous Byzantine consensus algorithms that provide low latency performance in favorable conditions while guaranteeing strong consistency when failures and contention occur;
- we prove lower bounds in the number of processors required for such algorithms;
- and we present Bosco, the first one-step algorithm for Byzantine consensus that meets these bounds.

3.1 The Byzantine Consensus Problem

The Byzantine consensus problem was first posed in [43], albeit for a synchronous environment. In this chapter we focus on an asynchronous environment.

In this problem, there is a set of n processors $P = \{p, q, \dots\}$ each of which have an initial value, 0 or 1. An unknown subset T of P contains *faulty* processors. These faulty processors may exhibit arbitrary (*aka* Byzantine) behavior, and may collude maliciously. Processors in $P - T$ are *correct* and behave according to some protocol. Processors communicate with each other by sending messages via a network. The network is assumed to be fully asynchronous but reliable, that is, messages may be arbitrarily delayed but between two correct processors, will

be eventually be delivered. Links between processors are private so a Byzantine processor cannot forge a message from a correct processor.

In addition, we assume a *strong network adversary*. By this, we mean that the network is controlled by an adversary that, with full knowledge of the contents of messages, may choose to arbitrarily delay messages as long as between any two correct processes, messages are eventually delivered.

The goal of a Byzantine consensus protocol is to allow all correct processors to eventually *decide* some value. Specifically, a protocol that solves Byzantine consensus must satisfy:

Definition: Agreement: If two correct processors decide, then they decide the same value. Also, if a correct processor decides more than once, it decides the same value each time.

Definition: Unanimity: If all correct processors have the same initial value v , then a correct processor that decides must decide v .

Definition: Validity: If a correct processor decides v , then v was the initial value of some processor.

Definition: Termination: All correct processors must eventually decide.

Note that algorithms that satisfy all of the above requirements are not possible in asynchronous environments when even a single crash failure must be tolerated [25]. In practice, algorithms circumvent this limitation by assuming some limitation in the extent of asynchrony in the system, or by relaxing the

Termination property to a probabilistic one where all correct processors terminate with probability 1.

Unanimity requires that the outcome be predetermined when the initial values of all correct processors are unanimous. A one-step algorithm takes advantage of such favorable initial conditions to allow correct processors to decide in one communication step.

We define two notions of one-step protocols:

Definition: Strongly one-step: If all correct processors have the same initial value v , a *strongly one-step* Byzantine consensus algorithm allows all correct processors to decide v in one communication step.

Definition: Weakly one-step: If there are no Byzantine processors in the system and all processors have the same initial value v , a *weakly one-step* Byzantine consensus algorithm allows all correct processors to decide v in one communication step.

While both can decide in one step, strongly one-step algorithms makes fewer assumptions about the required conditions and in particular cannot be slowed down by Byzantine failures when all correct processors have the same initial value. Strongly one-step algorithms optimize for the case where some processors may be faulty, but there is no contention among correct processors, and weakly one-step algorithms optimize for cases that are both contention-free *and* failure-free.

3.2 Lower Bounds

We show that a Byzantine consensus algorithm that tolerates t Byzantine failures among n processors requires $n > 7t$ to be strongly one-step and $n > 5t$ to be weakly one-step.¹ These results are for the best case scenario in which each correct processor broadcasts its initial value to all other processors in the first communication step and thus they hold for any algorithm.

3.2.1 Lower Bound for Strongly One-Step Byzantine Consensus

Lemma 7. *A strongly one-step Byzantine consensus algorithm must allow a correct processor to decide v after receiving the same initial value v from $n - 2t$ processors.*

Proof. Assume otherwise, that there exists a run in which a strongly one-step Byzantine algorithm \mathcal{A} does not allow a correct processor p to decide v after receiving the same initial value v from $n - 2t$ processors. Since \mathcal{A} is a strongly one-step algorithm, the fact that processor p does not decide after the first round implies that some correct processor q has an initial value v' , $v' \neq v$. Now consider a second run, in which all correct processors *do* have the same initial value v . Without blocking, p can wait for messages from at most $n - t$ processors. Among these, t can be Byzantine and send arbitrary initial values. This means that processor p is only guaranteed to receive $n - 2t$ messages indicating that $n - 2t$ processors have the initial value v . Given that \mathcal{A} is a strongly one-step al-

¹These results are for threshold quorum systems, but may be generalized to use arbitrary quorum systems.

gorithm, p must decide v at this point. However, from the point of view of p , this second run is indistinguishable from the first run. This is a contradiction. \square

Theorem 8. *Any strongly one-step Byzantine consensus protocol that tolerates t failures requires at least $7t + 1$ processors.*

Proof. Assume that there exists a strongly one-step Byzantine consensus algorithm \mathcal{A} that tolerates up to t Byzantine faults and requires only $7t$ processors. We divide the processors into three groups: G_0 and G_1 each contain $3t$ processors, of which the correct processors have initial values 0 and 1 respectively; G_* contain the remaining t processors.

Now consider the following configurations C_0 and C_1 . In C_0 , t of the processors in G_1 are Byzantine, and processors in G_* have the initial value 0. Assume that Byzantine processors act as if they are correct processors with initial value 0 when communicating with processors in G_* , and initial value 1 when communicating with processors not in G_* . Now consider that a correct processor $p_0 \in G_*$ collects messages from $n - t$ processors in the first communication step. Given that the network adversary controls the order of message delivery, p_0 can be made to receive messages from all processors in G_0 and G_* , and the t Byzantine processors in G_1 . p_0 thus receives $n - 2t$ messages indicating that the $n - 2t$ senders have initial value 0. By Lemma 7, p_0 must decide 0 after that first communication step. In order to satisfy Agreement, \mathcal{A} must ensure that any correct processor that ever decides in C_0 decides 0. We say that C_0 is 0-valent.

In C_1 , t of the processors in G_0 are Byzantine, and processors in G_* have the initial value 1. In addition, Byzantine processors act as if they are correct processors with initial value 1 when communicating with processors from G_* and initial value 0 when communicating with processors not in G_* . A correct

processor $p_1 \in G_*$ collects messages from $n - t$ in the first communication step. Suppose that the network adversary chooses to deliver messages from G_1 and G_* , as well as from the t Byzantine processors. Now p_1 collects $n - 2t$ messages indicating that $n - 2t$ senders have initial value 1. By Lemma 7, p_1 must decide 1 after the first communication step. In order to satisfy Agreement, \mathcal{A} must ensure that any correct processor that ever decides in C_1 decides 1. We say that C_1 is 1-valent.

Further assume that for both configurations, messages from any processor in G_* to any processor not in G_* are arbitrarily delayed such that in any asynchronous round, when a processor that is not in G_* awaits $n - t$ messages, it receives messages from every processor that is not in G_* . Now, any correct process $q_0 \notin G_*$ executing \mathcal{A} in C_0 will be communicating with $3t$ processors that behave as if they are correct processors with initial value 0 and $3t$ processors that behave as if they are correct processors with initial value 1. As we have shown above, C_0 is a 0-valent configuration, so \mathcal{A} must ensure that q_0 decides 0, if it ever decides. Similarly, a correct processor $q_1 \notin G_*$ executing \mathcal{A} in C_1 will also be communicating with $3t$ processors that behave as if they are correct processors with initial value 0 and $3t$ processors that behave as if they are correct processors with initial value 1. However, since we have shown that C_1 is a 1-valent configuration, \mathcal{A} must ensure that q_1 decides 1, even though it sees exactly the same inputs as q_0 . This is a contradiction. \square

3.2.2 Lower Bound for Weakly One-Step Byzantine Consensus

We now show the corresponding lower bound for weakly one-step algorithms. The lower bound for weakly one-step algorithms happens to be identical to that for two-step algorithms. The bound for two-step algorithms was shown in [49]. We show a corresponding bound for weakly one-step algorithm for completeness, but note that this is not a new result.

We weaken the requirement on Lemma 7 as follows:

Lemma 9. *A weakly one-step Byzantine consensus algorithm must allow a processor to decide v after learning that $n - t$ processors have the same initial value v .*

Proof. A processor can only wait for messages from $n - t$ processors without risking having to wait indefinitely. Since a weakly one-step Byzantine consensus algorithm must decide in one communication step if all correct processors have the same initial value and there are no Byzantine processors, it must decide if all of the $n - t$ messages claim the same initial value. \square

Theorem 10. *A weakly one-step Byzantine consensus protocol that tolerates t failures requires at least $5t + 1$ processors.*

Proof. We provide only a sketch of the proof since it is similar to that of Theorem 8. Proof by contradiction. Assume that a Byzantine consensus algorithm \mathcal{A} is weakly one-step and requires only $5t$ processors. We divide the $5t$ processors into three groups, G_0 , G_1 , and G_* , containing $2t$, $2t$, and t processors respectively. All correct processors in G_0 have the initial value 0 and all correct processors in G_1 have the initial value 1.

As in the proof of Theorem 8, we construct two configurations C_0 and C_1 . In C_0 , processors in G_* have the initial value 0 and t processors in G_1 are Byzantine. Correspondingly, in C_1 , processors in G_* have the initial value 1 and t processors in G_0 are Byzantine. These Byzantine processors behave as they do in the proof of Theorem 8. It is thus possible for processors in G_* to decide 0 and 1 in C_0 and C_1 respectively. Therefore, correct processors in G_0 and G_1 must not decide any value other than 0 and 1 respectively. However, if all messages from any processor in G_* to any processor not in G_* are delayed, then correct processors in C_0 and C_1 see exactly the same inputs. This is a contradiction. \square

3.3 Bosco

We now present Bosco (**B**yzantine **O**ne-**S**tep **C**onsensus), an algorithm that meets the bounds presented in the previous section. To the best of our knowledge, Bosco is the first strongly one-step algorithm that solves asynchronous Byzantine consensus with optimal resilience. The idea behind Bosco is simple, and resembles the one presented in [13]. We simply extend the results of [13] to handle Byzantine failures. The Bosco algorithm is shown in Algorithm 1.

Bosco is an asynchronous Byzantine consensus algorithm that satisfies Agreement, Unanimity, Validity, and Termination. Bosco requires $n > 3t$, where n is the number of processors in the system, and t is the maximum number of Byzantine failures that can be tolerated, in order to provide these correctness properties. In addition, Bosco is weakly one-step when $n > 5t$ and strongly one-step when $n > 7t$.

The main idea behind Bosco is that if all processors have the same initial

Algorithm 1: Bosco: a one-step asynchronous Byzantine consensus algorithm

Input: v_p

- 1 broadcast $\langle \text{VOTE}, v_p \rangle$ to all processors
 - 2 wait until $n - t$ VOTE messages have been received
 - 3 **if** more than $\frac{n+3t}{2}$ VOTE messages contain the same value v **then**
 - 4 DECIDE(v)
 - 5 **if** more than $\frac{n-t}{2}$ VOTE messages contain the same value v ,
 - 6 and there is only one such value v **then**
 - 7 $v_p \leftarrow v$
 - 8 Underlying-Consensus(v_p)
-

value, then given enough processors in the system, a correct processor is able to observe sufficient information to safely decide in the first communication round. Additional mechanisms ensure that if such an early decision ever happens, all correct processors *must* either i) early decide the same value; or ii) set their local estimates to the value that has been decided.

When the algorithm starts, each processor p receives an input value v_p , that is the value that the processor is trying to get decided and the value that it will use for its local estimate. Each processor broadcasts this initial value in a VOTE message, and then waits for VOTE messages from $n - t$ processors (likely including itself). Since at most t processors can fail, votes from $n - t$ processors will eventually be delivered to each correct processor.

Among the votes that are collected, each processor checks two thresholds: if more than $\frac{n+3t}{2}$ of the votes are for some value v , then a processor decides v ; if more than $\frac{n-t}{2}$ of the votes are for some value v , then a processor sets its

local estimate to v . Each processor then invokes `Underlying-Consensus`, a protocol that solves asynchronous Byzantine consensus (satisfies Agreement, Unanimity, Validity, and Termination), but is not necessarily one-step.

We first prove that `Bosco` satisfies Agreement, Unanimity, Validity, and Termination, when $n > 3t$.

Lemma 11. *If two correct processors p and q decide values v and v' in line 4, then $v = v'$.*

Proof. Assume otherwise, that two correct processors p and q decide values v and v' in line 4 such that $v \neq v'$. p and q must have collected more than $\frac{n+3t}{2}$ votes for v and v' each. Since there are only n processors in the system, these two sets of votes share more than $\frac{3t}{2}$ common senders. Given that only t of these senders can be Byzantine, more of $\frac{t}{2}$ of these senders are correct processors. Since a correct processor must send the same vote to all processors (in line 1), $v = v'$. This is a contradiction. \square

Lemma 12. *If a correct processor p decides a value v in line 4, then any correct processor q must set its local estimate to v in line 6.*

Proof. Assume otherwise, that a correct processor p decides a value v in line 4, and a correct processor q does not set its local estimate to v in line 6. Since processor p decides in line 4, it must have collected more than $\frac{n+3t}{2}$ votes for v in line 2. Since processor q does not set its local estimate to v in line 6, it must have collected no more than $\frac{n-t}{2}$ votes for v , or collected more than $\frac{n-t}{2}$ votes for some value v' , $v' \neq v$. For the first case, consider that since there are only n processors in the system, processor q must have collected votes from at least $n - 2t$ of the senders that processor p collected from. Among these, more than $\frac{n+t}{2}$ sent a vote

for v to q . Since at most t of these processors can be Byzantine, processor q must have received more than $\frac{n-t}{2}$ votes for v . This is a contradiction. For the second case, if q collects more than $\frac{n-t}{2}$ votes for some value v' , $v' \neq v$, then more than t of these senders must be among those that sent a vote for v to processor q . This is a contradiction, since, no more than t of the processors in the system can be Byzantine. \square

Theorem 13. *Bosco satisfies Agreement.*

Proof. There are two cases to consider. In the first case, no processor collects sufficient votes containing the same value to decide in line 4. This means that all decisions occur in Underlying-Consensus. Since Underlying-Consensus satisfies Agreement, Bosco satisfies Agreement. In the second case, some correct processor p decides some value v in line 4. By Lemma 11, any other processor that decides in line 4 must decide the same value. By Lemma 12, all correct processors must change their local estimates to v in line 6. Therefore, all correct processors will invoke Underlying-Consensus with the value v . Since Underlying-Consensus satisfies Unanimity, all correct processors that decide in Underlying-Consensus must also decide v . \square

Theorem 14. *Bosco satisfies Unanimity.*

Proof. Proof by contradiction. Suppose a processor p decides v' , but all correct processors have the same initial value v , $v' \neq v$. Since only t Byzantine processors can broadcast vote messages that contain $v \neq v'$, no correct processor can collect sufficient votes to either decide in line 4 or to set its local estimate in line 6. Therefore, in order for a processor to decide v , Underlying-Consensus must allow correct processors to decide v even though all correct processors start

Underlying-Consensus with the initial value v' . This is a contradiction since Underlying-Consensus satisfies Unanimity. \square

Theorem 15. *Bosco satisfies Validity.*

Proof. If a processor decides v in line 4, more than $\frac{n+3t}{2}$ processors voted v and more than $\frac{n+t}{2}$ of these processors are correct and had initial value v . Similarly, if a processor sets its local estimate to v in line 6, more than $\frac{n-t}{2}$ processors voted v and more than $\frac{n-3t}{2}$ of these processors are correct and had initial value v . Combined with the fact that Underlying-Consensus satisfies Validity, Bosco satisfies Validity. \square

Note that satisfying Validity in general in a consensus protocol is non-trivial, particularly if the range of initial values is large. A thorough examination of the hardness of satisfying Validity is beyond the scope of this chapter; we simply assume that Underlying-Consensus satisfies Validity for the range of initial values that it allows.

Theorem 16. *Bosco satisfies Termination.*

Proof. Since each processor awaits messages from $n - t$ processors in line 2, and there can only be t failures, line 2 is guaranteed not to block forever. Each processor will therefore invoke the underlying consensus protocol at some point. Therefore, Bosco inherits the Termination property of Underlying-Consensus. \square

Next, we show that Bosco offers strongly and weakly one-step properties when $n > 7t$ and $n > 5t$ respectively.

Theorem 17. *Bosco is Strongly One-Step if $n > 7t$.*

Proof. Assume that all correct processors have the same initial value v . Now consider any correct processor that collects $n - t$ votes in line 2. At most t of these votes can be from Byzantine processors and contain values other than v . Therefore, all correct processors must obtain at least $n - 2t$ votes for v . Since $n > 7t$, $2n - 4t > n + 3t$. This means that $n - 2t > \frac{n+3t}{2}$. Therefore, all correct processors will collect sufficient votes and decide in line 4. \square

Theorem 18. *Bosco is Weakly One-Step if $n > 5t$.*

Proof. Assume that there are no failures in the system and that all processors have the same initial value v . Then any correct processor must collect $n - t$ votes that contain v in line 2. Given that $n > 5t$, $2n - 2t > n + 3t$. This means that $n - t > \frac{n+3t}{2}$. Therefore, all correct processors will collect sufficient votes and decide in line 4. \square

3.4 RS-Bosco and WS-Bosco

One important feature of Bosco, from which it draws its simplicity, is its dependence on an underlying consensus protocol that it invokes as a subroutine. This allows the specification of Bosco to be free of complicated mechanisms typically found in consensus protocols to ensure correctness. While it is clear that any Byzantine fault-tolerant consensus protocol that provides Agreement, Unanimity, Validity, and Termination can be used for the subroutine in Bosco, the FLP impossibility result [25] states that such a protocol cannot actually exist! Two

common approaches have been used to sidestep the FLP result: assuming partial synchrony or relaxing the termination property to a probabilistic termination property. Thankfully, such algorithms can be used as subroutines to Bosco, resulting in one-step algorithms that either require partial synchrony assumptions, or provide probabilistic termination properties (or both). An example of an algorithm that can be used as a subroutine in Bosco is the Ben-Or algorithm [9]. Algorithms that do not provide validity, such as PBFT [15], cannot be used by Bosco.

While abstracting away the underlying consensus protocol simplifies the specification and correctness proof of Bosco, for practical purposes it may be advantageous to unroll the subroutine. This potentially allows piggybacking of messages and improves the efficiency of implementations. As an example, Algorithm 2 shows RS-Bosco, a randomized strongly one-step version of Bosco, and Algorithm 3 shows RW-Bosco, a randomized weakly one-step version of Bosco. Both RS-Bosco and RW-Bosco are not dependent on any underlying consensus algorithms.

RS-Bosco is strongly one-step and requires that $n > 7t$, while RW-Bosco is weakly one-step and requires that $n > 5t$. Both algorithms do not satisfy Termination as defined in section 3.1, but instead provides Probabilistic Termination:

Definition: Probabilistic Termination: All correct processors decide with probability 1.

We note that RS-Bosco suffers from two limitations as currently constructed. First, RS-Bosco solves only binary consensus. Second, RS-Bosco uses a local coin to randomly update local estimates when a threshold of identical votes cannot

be obtained. This mechanism is similar to that in the Ben-Or algorithm and causes the algorithm to require an exponential number of rounds for decision when contention is present. We believe that these limitations can be overcome in practical implementations.

We present the proof of correctness for RW-Bosco. The corresponding proofs for RS-Bosco are simpler and omitted for the sake of brevity.

Algorithm 2: RS-Bosco: a randomized strongly one-step asynchronous Byzantine consensus algorithm

```

1 Initialization
2    $x_p \leftarrow v_p$ 
3    $r_p \leftarrow 0$ 
4 Round  $r_p$ 
5   Broadcast  $\langle \text{VOTE}, r_p, x_p \rangle$  to all processors
6   Collect  $n - t$   $\langle \text{VOTE}, r_p, * \rangle$  messages
7   if more than  $\frac{n+3t}{2}$  VOTE msgs contain  $v$  then
8       DECIDE( $v$ )
9   if more than  $\frac{n-t}{2}$  VOTE msgs contain  $v$  then
10       Broadcast  $\langle \text{CANDIDATE}, r_p, v \rangle$ 
11   else
12       Broadcast  $\langle \text{CANDIDATE}, r_p, \perp \rangle$ 
13   end
14   Collect  $n - t$   $\langle \text{CANDIDATE}, r_p, * \rangle$  messages
15   if at least  $t + 1$  msgs are NOT of the form  $\langle \text{CANDIDATE}, r_p, x_p \rangle$  then
16        $x_p \leftarrow \text{RANDOM}()$  // pick randomly from  $\{0,1\}$ 
17    $r_p \leftarrow r_p + 1$ 

```

Algorithm 3: RW-Bosco: a randomized weakly one-step asynchronous Byzantine consensus algorithm

```

1 Initialization:  $x_p \leftarrow v_p, r_p \leftarrow 0$ 
2 Round  $r_p$ 
3   Broadcast  $\langle \text{VOTE}, r_p, x_p \rangle$  to all processors
4   Collect  $n - t$  messages of the form  $\langle \text{VOTE}, r_p, * \rangle$ 
5   if at least  $\lceil \frac{n+3t+1}{2} \rceil$  votes are of the form  $\langle \text{VOTE}, r_p, v \rangle$  then
6     DECIDE( $v$ )
7   if at least  $\lceil \frac{n-t+1}{2} \rceil$  votes are of the form  $\langle \text{VOTE}, r_p, v \rangle$  then
8     Broadcast  $\langle \text{CANDIDATE}, r_p, v \rangle$ 
9   else
10    Broadcast  $\langle \text{CANDIDATE}, r_p, \perp \rangle$ 
11    Collect  $n - t$  messages of the form  $\langle \text{CANDIDATE}, r_p, * \rangle$ 
12    if at least  $\lceil \frac{n+t+1}{2} \rceil$  VOTE msgs AND  $\lceil \frac{n+t+1}{2} \rceil$  CANDIDATE msgs contain  $v$  then
13      Broadcast  $\langle \text{SUGGEST}, r_p, v \rangle$ 
14    else
15      Broadcast  $\langle \text{SUGGEST}, r_p, \perp \rangle$ 
16      Collect  $n - t$  messages of the form  $\langle \text{SUGGEST}, r_p, * \rangle$ 
17      if at least  $\lceil \frac{n+t+1}{2} \rceil$  SUGGEST msgs contain  $v$  then
18        DECIDE( $v$ )
19      if at least  $t + 1$  SUGGEST msgs contain  $v$  then
20         $x_p \leftarrow v$ 
21      else if at least  $t + 1$  messages are NOT of the form  $\langle \text{CANDIDATE}, r_p, x_p \rangle$  then
22         $x_p \leftarrow \text{RANDOM}()$ 
23       $r_p \leftarrow r_p + 1$ 

```

Lemma 19. *If a processor decides a value v in line 6 of Algorithm 3, then at least $\lceil \frac{n+t+1}{2} \rceil$ correct processors have $x_p = v$ in the same round.*

Proof. A processor that decides a value v in line 6 must have received $\lceil \frac{n+3t+1}{2} \rceil$ votes for v (line 5). Of these votes, at most t can be from Byzantine processors. Hence at least $\lceil \frac{n+t+1}{2} \rceil$ of these votes must be from correct processors, and these correct processors must have $x_p = v$. \square

Lemma 20. *If at least $\lceil \frac{n+t+1}{2} \rceil$ correct processors have $x_p = v$ in some round r , then in any round $r' \geq r$, any processor p with an estimate $x_p = v$ will not change its x_p .*

Proof. If $\lceil \frac{n+t+1}{2} \rceil$ correct processors have $x_p = v$ in some round r , then all correct processors will receive at least $\lceil \frac{n-t+1}{2} \rceil$ votes of the form $\langle \text{VOTE}, r, v \rangle$ (line 7) and hence send $\langle \text{CANDIDATE}, r, v \rangle$ (line 10). Therefore, any processor p with $x_p = v$ will receive at most t messages (in line 15) that are not of the form $\langle \text{CANDIDATE}, r, v \rangle$ and hence not change its x_p value in line 22. A correct process will only change its value to v' in line 20 if it receives $t + 1$ SUGGEST messages containing the value v' . This means that at least one of the senders of those messages is correct. Since a correct process will only send a SUGGEST message containing value v' if it receives $\lceil \frac{n+t+1}{2} \rceil$ VOTE messages containing value v' , and $\lceil \frac{n+t+1}{2} \rceil$ correct processors have value $x_p = v$, it must be the case that $v = v'$. Therefore, in round $r + 1$, there will be, again, at least $\lceil \frac{n+t+1}{2} \rceil$ correct processors have $x_p = v$. By induction, all correct processors that have $x_p = v$ will not update their estimates in any round $r' \geq r$. \square

Lemma 21. *If a correct process decides a value v in line 18 in some round r , then all correct processes will set x_p to v in line 20 in round r .*

Proof. A correct process executes line 18 only if it receives $\lceil \frac{n+t+1}{2} \rceil$ SUGGEST messages containing v . This means that all other correct processes must receive at least $\lceil \frac{n-t+1}{2} \rceil$ SUGGEST. Since $n > 5t$, all processes must execute line 20 and set x_p to v . \square

Lemma 22. *If a correct process decides a value v in line 18 in some round r , then no correct process decided some value $v' \neq v$.*

Proof. By Lemma 21 and Lemma 20, if a process decides a value v in line 18, it cannot be the case that at least $\lceil \frac{n+t+1}{2} \rceil$ correct processes have $x_p = v'$ where $v' \neq v$. Therefore, by Lemma 19, it cannot be the case that some process decided some value v' in line 6 where $v' \neq v$. If two processes decide values v and v' in round r , then they must both have received $\lceil \frac{n+t+1}{2} \rceil$ SUGGEST messages containing the values v and v' respectively. At least one of the messages received by both processes must be from the same correct process, so $v = v'$. \square

Theorem 23. *Algorithm 3 satisfies Agreement.*

Proof. Agreement follows immediately Lemma 20, Lemma 19, and Lemma 22. \square

Lemma 24. *If all correct processors have the same local estimate v in round r , then all correct processors decide v in round r .*

Proof. If all correct processors have the same local estimate in some round r , then all correct processes must receive at least $\lceil \frac{n+t+1}{2} \rceil$ VOTE messages containing value v and send a corresponding CANDIDATE message containing value v in line 13. This will in turn cause all correct processes to send a SUGGEST message containing the same value. Therefore, all correct processes will receive at least $\lceil \frac{n+t+1}{2} \rceil$ SUGGEST messages containing value v and decide in line 18. \square

Theorem 25. *Algorithm 3 satisfies Unanimity and Validity.*

Proof. Unanimity follows immediately from Lemma 24. The algorithm provides an especially strong form of Validity in that it decides a value that was proposed by one of the *correct* processors. This follows directly from Unanimity and the fact that values are either 0 or 1. \square

Lemma 26. *In round r , if two correct processes p and q send *SUGGEST* messages in round r containing the values v and v' respectively, then $v = v'$.*

Proof. Processes p and q send *SUGGEST* messages because they received at least $\lceil \frac{n+t+1}{2} \rceil$ *VOTE* messages containing v and v' respectively. Since there are at most t Byzantine processes, at least one of the messages received by both processes were from the same correct sender. Hence, $v = v'$. \square

Lemma 27. *In round r , if two processes p and q update their local estimates x_p and x_q to v and v' respectively in line 20, then $v = v'$.*

Proof. Processes p and q each received at least $t + 1$ *SUGGEST* messages containing the values v and v' respectively. At least one of the messages received by p and one of the messages received by q must have been sent by correct senders. By Lemma 26, they must contain the same value. \square

Lemma 28. *In a round r , if a correct process p sets $x_p \leftarrow v$ in line 20, then any correct q with $x_q \neq v$ that does not update x_q in line 20 must update x_q in line 22.*

Proof. Process p updates x_p in line 20 because it receives $t+1$ *SUGGEST* messages, one of which must be from an honest sender. The honest sender must have received $\lceil \frac{n+t+1}{2} \rceil$ *CANDIDATE* messages containing value v . This means that q

must receive at least $\lceil \frac{n-3t+1}{2} \rceil$ CANDIDATE messages containing value v and hence execute line 22 if it does not execute line 20. \square

Lemma 29. *In some round r , if there are two correct processors p and q such that $x_p \neq x_q$, then either p or q will update its estimate.*

Proof. Assume otherwise. In a round r , there exists two correct processors p and q such that $x_p \neq x_q$ and neither p nor q executes line 20 or line 22. Without loss of generality, assume that $x_p = 0$ and $x_q = 1$. Since p does not execute line 22, at least $n - 2t$ of the CANDIDATE messages collected by p in line 11 are of the form $\langle \text{CANDIDATE}, r, 0 \rangle$, $n - 3t$ of which must have been sent by correct processors. Given that q collects $n - t$ messages, q must have received at least $n - 4t$ $\langle \text{CANDIDATE}, r, 0 \rangle$ messages. Since $n > 5t$, the equality test in line 12 must hold for q and thus q must have executed line 22 if it did not execute line 20. This is a contradiction. \square

Theorem 30. *Algorithm 3 satisfies Non-Blocking.*

Proof. Non-Blocking holds because in the absence of unanimity, Lemma 29 says that either all processes with a 0 estimate update their estimates; or all processes with a 1 estimate update their estimates; or both. If, by non-zero chance, all correct processors update their estimate to the same value, then in the next round, a decision is guaranteed by Lemma 24. \square

Theorem 31. *Algorithm 3 is weakly one-step if $n > 5t$.*

Proof. If all correct processors have the same initial value v and there are no Byzantine faults, then the number of votes containing the value v that are collected in line 4 in the first communication step must be at least $n - t$. If $n > 5t$,

then $2n - 2t > n + 3t$, and thus $n - t \geq \lceil \frac{n+3t+1}{2} \rceil$. Therefore, all correct processors must decide in the first communication step. \square

CHAPTER 4

K-SET AGREEMENT IN ONE COMMUNICATION STEP

Bosco was a one-step algorithm for a generalization of the consensus problem, where faulty processors can exhibit Byzantine failures in addition to crash failures. The consensus problem can also be generalized in a different way. Chaudhuri [21] introduced the k -set agreement problem where she relaxes the requirement that a single unique value is decided by all correct processes. As in consensus, processes propose values and each correct process has to decide a value. However, in k -set agreement, rather than requiring that only a single unique value is decided, up to k distinct values can be decided. In this chapter, we look at the one-step algorithms in the context of this generalized version of consensus.

We make two main contributions. First, we present the first algorithm for one-step k -set agreement. While every consensus algorithm is a k -set agreement algorithm, Keidar and Rajsbaum [39] showed that any consensus algorithm must have a run that requires two communication steps even when proposed values are restricted to $\{0, 1\}$. Thus no consensus algorithm is a one-step k -set agreement algorithm, where $k > 1$. Second, while we do not prove a general lower bound, we show that our algorithm is optimal for a natural class of algorithms, characterizing how any better algorithm must differ.

Chaudhuri [21] proposed the k -set agreement problem to investigate whether increasing the number of choices available to processors makes the consensus problem easier. She showed that k -set agreement is indeed easier, in the sense that correct algorithms exist that can tolerate $k - 1$ faults while the FLP result shows that consensus algorithms can only tolerate $1 - 1 = 0$ faults.

In the same spirit, our investigation of one-step k -set agreement shows that it is both easier and (possibly) harder than consensus. It is easier, in the sense that there are more initial configurations that allow processors to decide in a single round. For our algorithm, it is harder in the sense that the number of processors needed to guarantee correctness is larger for most $k > 1$ than for $k = 1$. Whether it is truly harder in this sense for all such algorithms remains an open problem.

4.1 k -Set Agreement

We assume a communication model as described in [25, 17]. We consider a finite set Π of n processors, namely, $\Pi = \{p_1, \dots, p_n\}$. Processors communicate by sending messages to one another via reliable communication channels. Processors and the network are fully asynchronous, that is, no assumptions are made about the relative speed of processors, or the time it takes for a message to be delivered. Up to f processors can fail by *crashing*, and a processor that crashes stops sending messages and never recovers.

In the k -set agreement problem, each processor *proposes* a value v_i , also called its *initial value*, and all correct processes have to *decide* on some value v . The objective of a k -set agreement protocol is to coordinate the decision of values such that the properties of *Validity*, *Termination*, and *k -Agreement* hold. Validity and Termination are defined in Section 3.1. *k -Agreement* is defined as follows:

Definition: k -Agreement: The set of values on which correct processors decide has size at most k .

The difficulty of the k -set agreement problem depends on the relationship of k , the maximum size of the set of distinct decided values, and f , the maximum

number of failures. If $f < k$, the problem is trivially solvable for any n [20]. For $f \geq k$, a generalization of the FLP result shows that the problem cannot be solved if termination is required in all runs (rather than with probability 1) [12, 35]. We follow the approach of Mostefaoui and Raynal [58] who showed that the problem can be solved by a randomized algorithm for $n > 2f + 1$. Alternative approaches to relaxing the termination condition include using failure detectors or establishing conditions that restrict the inputs to the problem [54, 55].

In addition, it is possible and desirable for protocols to allow processors to decide in a single communication step when there is little contention among the processors. More precisely, we define the following:

Definition: One-Step: If the set of initial values of processors has size at most k then a one-step k -set agreement algorithm allows all correct processors to decide in one communication step.

We call an algorithm that satisfies k -Agreement, Validity, Termination, and One-Step a *one-step k -set agreement* algorithm. Note that the trivial algorithm for $f < k$ is a one-step algorithm; for the remainder of the chapter, we ignore the trivial case and assume $f \geq k$.

Assumption 1. $f \geq k$

4.2 Lower Bounds

We will now define a class of k -set agreement protocols, namely *popularity-based* protocols, and show that any popularity-based one-step k -set agreement proto-

col must require more than $fk - k^2 + f + \max(f + 1, 2k)$ processors in order to tolerate f crashes.

Lemma 32. *A one-step k -set agreement protocol must allow a correct processor to decide after receiving no more than k distinct initial values from $n - f$ processors.*

Proof. Assume otherwise. Then there exists a run where a correct algorithm \mathcal{A} does not allow a processor p with initial value v_p to decide after receiving at most k different initial values from $n - f$ processors. Since \mathcal{A} is one-step and p does not decide, there exist at least $k + 1$ distinct values, i.e., there exists some processor q with initial value v' that p did not receive in the first round. Now consider a second run where p again receives at most k different initial values from $n - f$ processors, but where the remaining f processors have the same initial value v_p as p . Without blocking, p can wait for messages from at most $n - f$ processors. Thus, p must decide at this point. However, from the point of view of p , this second run is indistinguishable from the first run. This is a contradiction.

□

An insight from Lemma 32 is that any correct algorithm can be expressed in the form shown in Algorithm 4. After receiving at most k distinct values among $n - f$ initial values, a correct processor must use some rule to decide on one of them. A processor that does not decide in this step continues as specified by the algorithm.

A natural decision rule is for the processor to decide the value for which it received the most copies in step A2 (breaking ties using some ordering over values). We call a one-step k -set agreement algorithm using such a decision rule a

Algorithm 4: The general form for any one-step k -set agreement protocol

Input: v_p

- A1 broadcast $\langle v_p \rangle$ to all processors
 - A2 wait until $n - f$ messages have been received
 - A3 **if** *the messages contain at most k values* **then**
 - A4 decide using DECISION RULE
 - A5 rest of algorithm
-

popularity-based algorithm, and give a lower bound on the number of processors required for such algorithms. This does not rule out the possibility of a better algorithm that uses a different decision rule.

Lemma 33. *Any popularity-based algorithm requires $n > fk - k^2 + 2f + 1$.*

We prove Lemma 33 by demonstrating a pair of runs in which an algorithm is forced to make an inconsistent decision with lower values of n . To provide insight before presenting the proof, we analyze the specific case of $k = 1$, where the bound is $n > 3f$. Note that this is equivalent to the bound for one-step consensus shown in [13].

Consider a run of $3f$ processors using some popularity-based algorithm \mathcal{A} . We divide the processors into three groups: G_* contains f processors that crash immediately; G_1 contains f processors that have initial value v_1 ; and G_2 contains f processors that have initial value v_2 . Given the Termination and k -Agreement property of \mathcal{A} , these processors must eventually decide a single value. WLOG, let them decide v_1 .

Now consider a run where every processor in G_* has value v_2 . $f - 1$ of the processors send their value to the remaining processor p and then crash. p re-

ceives f messages from processors in G_* and f messages from processors in G_2 , all with value v_2 , so it decides v_2 in step A5. No other processor receives a message from p . To all processors in G_1 and G_2 , this run is indistinguishable from the previous run, so they decide v_1 violating 1-Agreement. We now construct the proof by generalizing this example.

Proof. Consider a run of a popularity-based algorithm \mathcal{A} with $n = fk - k^2 + 2f + 1$ processors. We divide the processors into $k + 2$ groups, namely, G_1, \dots, G_{k+1} , and G_* . Each group G_i in $\{G_1, \dots, G_{k+1}\}$ contains $f - k + 1$ processors that have initial value v_i , for a total of $fk + f - k^2 + 1$ processors. The remaining f processors are in G_* and crash immediately. Each correct processor (processors not in G_*) must eventually decide a value and the total number of decided values must not exceed k . WLOG, let at least one processor decide v_{k+1} .

Now consider a second run that is identical except that each processor $p_i \in G_* = \{p_1, \dots, p_f\}$ has initial value $v_{i \bmod k}$. In step 2, each processor p_i in G_* receives messages from all processors except all the processors in G_{k+1} and one processor in each of G_j where $1 \leq j \leq k, j \neq i$. WLOG, assume the decision rule breaks ties for the most popular value in step A4 by deciding the value v_i with the largest i . Then each processor $p_i \in G_*$ decides its own value v_i in step 4, by virtue of collecting more votes for v_i than any other value, or by breaking ties in favor of v_i . Hence, every value v_1, \dots, v_k is decided by some processor. The other $n - f$ processors that are not in G_* receive no messages from these f processors, so this run is indistinguishable from the previous run. Thus at least one processor decides v_{k+1} , contradicting the k -Agreement property of the algorithm. \square

Lemma 34. *Any popularity-based algorithm requires $n > fk - k^2 + f + 2k$.*

Again, for illustration, consider the case of $k = 3$ and $f = 4$. For these values of k and f , Lemma 33 requires $n > 12$. We show that this is not sufficient. Consider $n = 13$. Again, WLOG, we assume that the decision rule breaks ties by deciding the greatest value among the most popular. Consider a run where the processors are divided into four groups: G_1, G_2, G_3 , and G_4 . G_1 contains four processors that have initial value v_1 , and each of G_2, G_3 , and G_4 contain three nodes, with initial values v_2, v_3 , and v_4 respectively. It is not difficult to see that there exists a run where the message delivery order causes two processors in G_1 and one processor in each of G_3 and G_4 to decide their initial value after the first communication step. We refer to these early deciding processors as G_* . Now assume that messages from processors in G_* to all other processors are arbitrarily delayed, such that other processors do not receive any messages from G_* before deciding.

Given that processors in G_* have decided values v_1, v_3 , and v_4 , and $k = 3$, all other processors must decide one of these values in order to satisfy k -Agreement. WLOG, suppose some processor that's not in G_* decides v_4 . Now consider another run, in which the processor in G_* that decided v_4 instead has value v_2 and decides v_2 . To all processors not in G_* , this run is indistinguishable from the first, so some processor must decide v_4 . This violates k -Agreement and is a contradiction.

Proof. Consider a run with $n = fk - k^2 + f + 2k$. We divide the processors into $k + 1$ groups, namely G_1, \dots, G_{k+1} where processors in each group G_i have initial value v_i . Each of the first $k - 2$ groups have $f - k + 3$ processors, and all the other groups (G_{k-1}, G_k , and G_{k+1}) have $f - k + 2$ processors.

Now consider any processor in groups $G_1 \dots G_{k-2}$. The order of message de-

livery can occur in such a way that the processor decides its own initial value in step A4: the processor receives all messages except from all the processors in G_{k+1} , and a single processor in each of the groups $G_1 \dots G_{k-2}$ that it does not belong to. Since the processor sees no more than k different values in step A2 and its own initial value is the most common, it decides its initial value. Similarly, depending on the order of message delivery, it is possible for a processor in groups G_k and G_{k+1} to decide its own initial value in step A4. Consider a processor p in G_k . If in the first round, it receives all messages from all processors except G_{k+1} and one processor from each of $G_1 \dots G_{k-2}$, then it would have seen exactly k different values. Based on the tie breaking rule, it decides k . A processor in G_{k+1} can decide v_{k+1} based on a similar construction. Note that given our assignment of initial values, processors in G_{k-1} cannot decide v_{k-1} in step A4 regardless of the order of message delivery.

Now consider a run where one processor in each of $G_1 \dots G_{k+1}$ other than G_{k-1} decides its initial value in in step A4. We refer to these processors as G_* . Assume that all messages from processors in G_* to processors not in G_* are arbitrarily delayed so that processors not in G_* never receive any of those messages before deciding. Since k different values ($v_i, 1 \leq i \leq k-2, k, k+1$) have been decided in step A4 by processors in G_* , all processors that are not in G_* must eventually decide one of these values. Let one of these processors, p_α , decide v_α , where $v_\alpha \neq v_{k-1}$.

Now consider a second run, which is identical in every way to the run we just described, except that the one processor in G_* which decided v_α now instead has initial value v_{k-1} , and decides v_{k-1} in step A4. This run is indistinguishable from the previous run to all processors not in G_* , so p_α must eventually decide

v_α . This violates k -Agreement and is a contradiction.

□

Combining Lemma 33 and Lemma 34 gives our lower bound.

Theorem 35. *Any popularity-based algorithm requires $n > fk - k^2 + f + \max(f + 1, 2k)$.*

4.3 Algorithm

Using the form presented in Section 4.2, we present one-step k -set agreement algorithm that meets lower bound stated by Theorem 35. The algorithm is shown here:

Algorithm 5: A one-step asynchronous k -set agreement protocol

Input: v_p

B1 broadcast $\langle v_p \rangle$ to all processors

B2 wait until $n - f$ messages have been received

B3 $V_p \leftarrow$ highest value among those received in the most messages

B4 *if the messages contain at most k values* **then**

B5 DECIDE(V_p)

B6 Underlying-Consensus(V_p)

The algorithm differs from the skeleton described in Section 4.2 in two important ways. First, we flesh out the decision rule that is used to decide a value. In particular, as in all popularity-based algorithms, each node collects $n - f$ messages and count the number of messages that contain each distinct value, and decide on the most popular one. If there is a tie between multiple values, the

greatest value is used. Second, after attempting to decide in the first round, each node then proceeds to run some protocol that solves the standard agreement problem. This can be any standard agreement protocol, with the one caveat being that the protocol must solve multi-value consensus (as opposed to binary consensus) and satisfy Validity, as defined in Section 4.1. For simplicity, we assume that the underlying consensus algorithm relies on randomization and uses no failure detectors, but the algorithm is also correct if failure detector-based algorithms are used.

We now prove the correctness of the algorithm.

Assumption 2. $n > fk - k^2 + f + \max(2k, f + 1)$

Assumption 3. *The underlying consensus protocol satisfies Validity, 1-Agreement, and Termination for $n > fk - k^2 + f + \max(2k, f + 1) \geq 3f$.*

Theorem 36. *Algorithm 5 satisfies Validity*

Proof. If a processor decides in step B5, it decides on some value v_p it received in a message $\langle v_p \rangle$. Otherwise it decides in step B6. Since the underlying consensus protocol satisfies Validity, it decides on the value v'_p of some processor p' . p' received this value in some message $\langle v'_p \rangle$. In either case, it decides on the value sent in some message, which is the input of some processor. \square

To show agreement, we first consider only runs where there are exactly $k + 1$ distinct initial values. For such runs, let x be the number of processors that start with the least popular value.

Lemma 37. *Algorithm 5 satisfies k -Agreement in runs with $k + 1$ initial values where $x > f - k + 2$.*

Proof. There is at least one value (the least popular) with which x processors start. If in step B2 a processor receives messages from all but the x processors with some value that only appears x times, it will decide the most popular value in step 5. In order to decide some other value, it must not receive a message from at least one processor with the most popular value, in which case it will decide the second most popular value. Since all x messages of a particular value must not be received by a processor that decides in step B5, such a processor can only miss messages containing $f - x$ other distinct values. Thus, at most the $f - x + 1 \leq f - (f - k + 3) + 1 = k - 2$ most popular values can be decided in this fashion.

If the $k - 2$ most popular values can be decided and the $(k - 2)$ th most popular value appears exactly x times, then the $(k - 1)$ th most popular value can also be decided by having the value for which no messages are received be the $(k - 2)$ th most popular value. This requires using all f failures, so a maximum of $k - 1$ distinct values can be decided in step B5. By the 1-Agreement property of the underlying consensus protocol, only one value will be decided in step B6. Therefore, at most k values will be decided in total. \square

Lemma 38. *Algorithm 5 satisfies k -Agreement in runs with $k + 1$ initial values where $x = f - k + 2$.*

Proof. If the $(k - 1)$ th most popular value appears more than x times, then the argument in the proof of Lemma 37 shows that at most $k - 1$ values will be decided in step B5 and so k -Agreement is satisfied.

Suppose that the $(k - 1)$ th most popular value appears exactly x times. By Assumption 2, $n > fk - k^2 + f + 2k$. Since $x = f - k + 2$, $n - x(k + 1) > k - 2$. At least x processors have each value and the $(k - 1)$ th, k th and $(k + 1)$ th most

popular values appear exactly x times, so the $k - 2$ most popular values must appear at least a total of $(k - 2)x + k - 1$ times. If a processor decides a value that is not one of the $k - 2$ most popular, it must have received at most x messages from processors with each of these values. Thus it must not receive at least $k - 1$ messages from such processors and a further $x = f - k + 2$ messages so that it does not receive more than k distinct values. This means it did not receive a total of $f + 1$ messages, which is not possible. Thus in this case at most $k - 2$ values can be decided and Algorithm 5 satisfies k -Agreement.

□

Lemma 39. *Algorithm 5 satisfies k -Agreement in runs with $k + 1$ initial values where $x < f - k + 2$.*

Proof. By Assumption 2, $n > fk - k^2 + f + (f + 1)$. Since $x < f - k + 2$, $n > xk + f + x$. x processors start with the least popular value, so a total of at least $xk + f + 1$ processors start with the remaining k values. Every processor must receive at least $xk + f + 1 - f = xk + 1$ messages from these processors in step B4 and thus receive at least $x + 1$ messages for one of the k most popular values. Thus in step B3, V_p will be set to one of these k values for each processor. Any decided value was set in step B3 by some processor, so Algorithm 5 satisfies k -Agreement. □

Theorem 40. *Algorithm 5 satisfies k -Agreement*

Proof. Lemmas 37, 38, and 39 prove k -Agreement for runs where there are initially exactly $k + 1$ distinct initial values. Runs with less than $k + 1$ initial values trivially satisfy k -Agreement because Algorithm 5 satisfies Validity (Theorem 4.3).

Consider a run with more than $k + 1$ initial values that violates k -Agreement. We complete the proof by showing that there must exist a run with $k + 1$ initial values that violates k -Agreement.

If a run violates k -Agreement then either (1) at least $k + 1$ values are decided in step B5 or (2) k values are decided in step B5 and some processor p set V_p to a value not among those k in step B3. In the first case, let S be the set consisting of $k + 1$ values that are decided in step B5 and let v_\perp be the least popular among them (breaking ties by selecting the greatest value). Now consider the run where each processor p has value $v'_p = v_p$ if $v_p \in S$ and $v'_p = v_\perp$ otherwise. There exists an order of message deliveries that will cause the same $k + 1$ values to be decided in the second run. In particular, a process that decides must receive messages with only k distinct values. In the first run, either each process that decided on a value in S received no messages from any processor with a value not in S or it does not receive any messages with at least two of the values in S (because it received a message with some other value). These messages can be replaced by messages containing values in S . Thus there is a run which violates k -Agreement and has only $k + 1$ initial values.

For the second case, we use a similar construction. Now S is the set of k values decided in step B5 and v_\perp is the value some processor p set V_p to in step B3 that is not in S . Define the initial values for the second run as before. By the same argument, there is an ordering of message deliveries that will cause the k distinct values to still be decided in step B5. Furthermore, p cannot receive fewer messages with value v_\perp in the second run than it did in the first run, so the value it computes in step B3 will still be v_\perp . Thus there is a run which violates k -Agreement and has only $k + 1$ initial values. \square

Theorem 41. *Algorithm 5 satisfies Termination*

Proof. Since each processor awaits messages from $n-f$ processors in step B2, and there can only be f failures, step B2 is guaranteed not to block forever. Each processor will therefore invoke the underlying consensus protocol at some point. Therefore, Algorithm 5 inherits the Termination property of the underlying consensus protocol. \square

Theorem 42. *Algorithm 5 satisfies One-Step*

Proof. Suppose the processors have at most k different initial values. Then in step B2, each processor will receive at most k different values. Thus each processor will decide in step B5. \square

CHAPTER 5

RPC CHAINS: EFFICIENT CLIENT-SERVER COMMUNICATION IN GEODISTRIBUTED SYSTEMS

Distributed enterprise applications, such as web applications, are often built from more basic services, such as storage services, database management systems, authentication and configuration services, and services for interfacing with external components (e.g., credit card processing, banking, vendors, etc). As systems become larger, more complex, and more ubiquitous, there is a corresponding increase in the number, diversity, and geographical dispersion of the remote services that they use. For instance, Hotmail and Live Messenger share an address book service and an authentication service; there are also services specialized for each application, say, for email storage or virus scanning. These services are heterogeneous; they are often developed by different teams and are *geo-distributed*, running in different parts of the world.

Geo-distribution provides many benefits: high availability, disaster tolerance, locality, and ability to scale beyond one data center or site. However, the thin and slow links connecting different sites pose challenges, especially in an enterprise setting, where applications have strict performance requirements. For instance, web applications should ideally respond within one second [60].

The most common primitives for inter-service communication are remote procedure calls (RPC's) or RPC-like mechanisms. RPC's can impose undesirable communication patterns and overheads when a client needs to make multiple calls to servers. This is because RPC's impose communication of the form $A-B-A$ (A calls B which returns to A) even though this pattern may not be optimal. For example, in Figure 5.1 left, a client A in site 1 uses RPC's to consec-

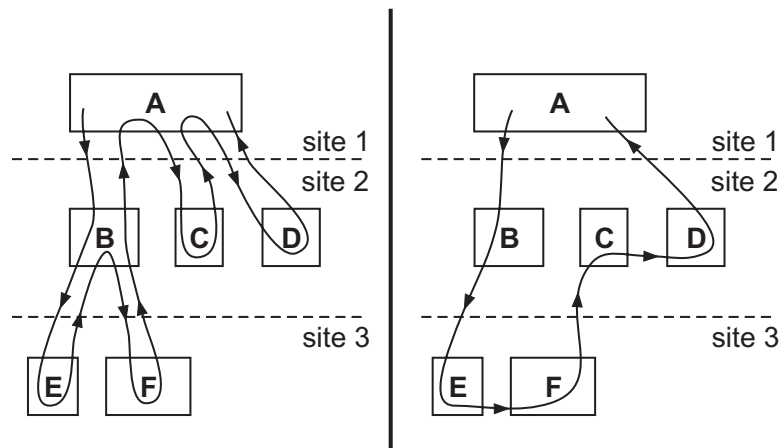


Figure 5.1: Standard RPCs vs an RPC chain

utively call servers B , C , and D in site 2. Server B , in turn, calls servers E and F in site 3. The use of RPC's forces the execution to return to A and B multiple times, causing 10 crossings of inter-site links

We propose a simple but more general communication primitive called a *Chain of Remote Procedure Calls*, or simply *RPC chain*, which allows a client to call multiple servers in succession ($A-B_1-B_2-\dots-A$), where the request flows from server to server without involving the client every time. The result is a much improved communication pattern, with fewer communication hops, lower end-to-end latency, and often lower bandwidth consumption. In Figure 5.1, we see how an RPC chain reduces the number of inter-site crossings to 4. The example in this figure is representative of a web mail application, where host A is a web server that retrieves a message from an email server B , then retrieves an associated calendar entry from a calendar service C , and finally retrieves relevant ads from an ad server D .

The key idea of RPC chains is to embed the chaining logic as part of the RPC call. This logic can be a generic function, constrained by some simple isolation

mechanisms. RPC chains have three important features:

- (1) *Server modularity*. What made RPC's so successful is the clean decoupling of server code, which allows servers to be developed independently of each other and the client. RPC chains preserve this attribute, even allowing existing legacy RPC's to be part of a chain through simple wrappers.
- (2) *Chain composability*. If a server in the chain itself wishes to call another server, this nested call can be simply added to the chain in flux. In Figure 5.1, when client *A* starts the chain, it intends to call only servers *B*, *C*, and *D*. But server *B* wants to call servers *E* and *F*, and so it adds them to the chain.
- (3) *Chain dynamicity*. The services that a host calls need not be defined a priori; they can vary dynamically during execution. In the left figure, the fact that client *A* calls servers *C* and *D* need not be known before *A* calls server *B*; it can depend on the result returned by *B*. For example, an error condition may cause a chain to end immediately instead of continuing on to the next server.

We demonstrate RPC chains through a storage and a web application. For the storage application, we show how a storage server can be enabled to use RPC chains, and we give a simple use in which a client can copy data between servers without having to handle the data itself. This speeds up the copying and saves significant bandwidth. For the web application, we implement a simple web mail service that uses chains to reduce the overheads of an ad server.

5.1 Setting

We consider enterprise systems that span geographically-diverse sites, where each site is a local area network. Sites are connected to each other through thinner and slower wide area links. Wide-area links can be made faster by improving the underlying network, and lots of progress has been made here, but this progress is hindered by economic barriers (e.g., legacy infrastructure), technological obstacles (e.g., switching speeds), and fundamental physical limitations (e.g., speed of light). Thus, the large discrepancy between the performance of local and wide-area links will continue.

Unlike the Internet as a whole, enterprise systems operate in a trusted environment with a single administrative domain and experience little churn. These systems may contain a wide range of services, often developed by many different teams, including general services for storage, database management, authentication, and directories, as well as application-specific services, such as email spam detection, address book management, and advertising. These services are often accessed using RPC's, which we broadly define as a mechanism in which a client sends a request to a server and the server sends back a reply. This definition includes many types of client-server interactions, such as the interactions in CORBA, COM, REST, SOAP, etc.

In enterprise environments, application developers are not malicious though some level of isolation is desirable so that a problem in one application or service does not affect others.

5.2 Design

5.2.1 Main mechanism

Servers provide services in the form of *service functions*, which is the general term we use for remote procedures, remote methods, or any other processing units at servers. An RPC chain calls a sequence of service functions, possibly at different servers. Service functions are connected together via *chaining functions*, which specify the next service function to execute in a chain (see Figure 5.2 top). Chaining functions are provided by the client and executed at the server. They can be arbitrary C# methods with the restriction that they be *stand-alone* code, that is, code which does not refer to non-local variables and functions, so that they can be compiled by themselves.

We chose this general form of chaining for two reasons. First, we want to allow the chain to unfold dynamically, so that the choice of next hop depends on what happens earlier in the chain. For example, an error at a service function could shorten a chain. Second, we wanted to support server modularity, so that services and client applications can be developed independently. Thus, a server may not produce output that is immediately ready for another server, in the way intended by the client's application. One may need to convert formats, reorder parameters, combine them, or even combine the outputs from several servers in the chain. For example, an NFS server does not output data in the format expected by a SQL server: one needs glue that will convert the output, choose the tables, and add the appropriate SQL wrapper, according to application needs. Chaining functions provide this glue. We initially considered a simpler alternative to chaining functions, in which a client just provides a static list of servers

```

// service function
object sf(object parmlist)
  // parmlist: parameter list

// chaining function
nexthop cf(object state, object result)
  // state: from client or earlier parts of chain
  // result: from last preceding service function
  // returns next chain hop:
  //      (server, sf_name, parmlist,
  //      cf_name, state)

```

```

chain_id start_chain(machine_t server,
  string sf_name, object parmlist,
  string cf_name, object state)

```

Figure 5.2: Signature of a service function and chaining function; signature of a function that initiates an RPC chain.

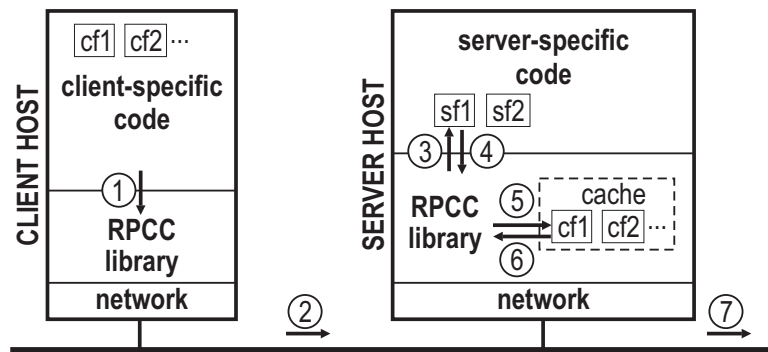


Figure 5.3: Execution of an RPC chain.

to call, but this design does not address the issues above. We also note that it is easy to translate a static server list into the appropriate chaining functions (one could even write a programmer tool that automatically does that), so our design includes static lists as a special case.

Figure 5.3 shows how an RPC chain executes. (1) A client calls our RPCC (RPC chain) library, specifying a server, a reference to a service function sf_1 at that server, its parameters, and a chaining function cf_1 . (2) This information is

then sent to the chosen server. (3) The server executes service function sf_1 , which (4) returns a result. (5) This result is passed to the chaining function cf_1 , which then (6) returns the next server, service function, and chaining function, and (7) the chain continues.

For example, suppose client A wants to call service functions sf_B, sf_C, sf_D at servers B, C, and D, in this order. To do so, the client specifies a reference to sf_B and a chaining function cf_1 . cf_1 causes a call to sf_C at server C with a chaining function cf_2 , which in turn causes a call to sf_D at server D with a chaining function cf_3 , which causes the final result to be returned to the client A.

5.2.2 Chaining function repository

Chaining functions are provided by clients but executed at servers. To save bandwidth, in our implementation the client does not send the actual code to the server. Rather, the client uploads the code to a repository, and sends a reference to the server; the server downloads the code from the repository and caches it for subsequent use. The repository stores chaining functions in source code format, which is then compiled by servers at runtime using the reflection capabilities of .NET/C# (Java has similar capabilities).

Storing source code introduces fewer dependencies, is more robust (binary formats change more frequently), and simplifies debugging. Because the cost of runtime compilation can be significant (≈ 50 ms, see Section 5.4.2), servers cache the compiled code, not the source code, to avoid repeated compilations.

When the chaining function is very small, it can be transmitted by the client

with the RPC chain, so that the server does not have to contact the repository.

5.2.3 Parameters and state

A chaining function is client logic that may depend on run-time variables, tables, or other state from the client or earlier parts of the chain. This state needs to be passed along the chain, and should ideally be small, otherwise its transmission cost can outweigh the benefits of an RPC chain (see Section 5.4.2). We represent the state as a set of name-value pairs, which is passed as a parameter to the chaining function (see Figure 5.2).

The output of each service function is also passed as a parameter to the subsequent chaining function. For example, in our storage copy application (Section 5.3.1), the first service function reads a file, and the chaining function uses the result as input to the next service function, which writes to a file on a different server. In our email application, a service function reads an email message, and the chaining function adds the message to the state of the next chaining function, so that the message is passed along the chain back to the chain originator (a mail web server).

5.2.4 Nesting and composition

RPC chains can be nested: a service function in a chain may itself start a sub-chain. For example, the main chain could call a storage service, which then needs to call a replica. We implement nesting so that a nested chain can be adjoined to an existing chain, as shown in Figure 5.4. The left side of the figure

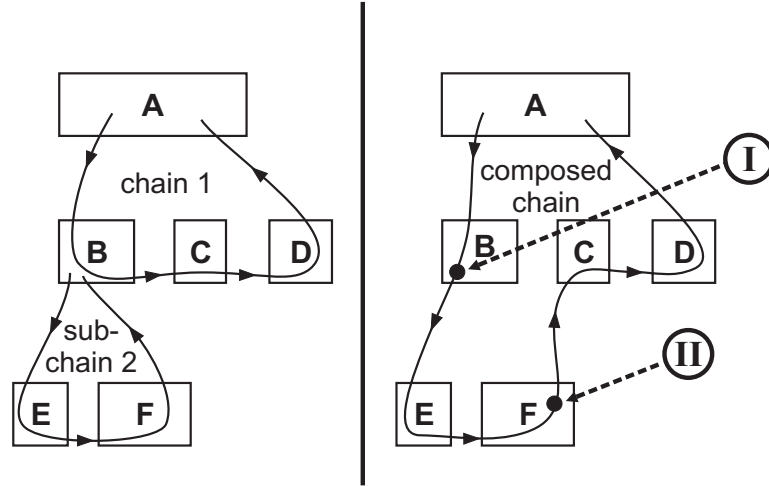


Figure 5.4: Composition of nested chains.

shows two chains, a main chain 1 and a sub-chain 2. On the right, the two chains are composed together. (I) marks where B starts a sub-chain, and (II) shows where the sub-chain ends and the main chain resumes.

Note the difference between starting a chain going from B to E, and moving to the next host in a chain going from C to D: the former occurs when the *service function* at B starts a new chain, while the latter occurs when the *chaining function* at C calls the next node in the chain. This distinction is important because the service function at B represents a native procedure at the service, while a chaining function at C represents logic coming from A. At E, the chaining function that calls F represents logic coming from B.

To compose a chain with its sub-chain, the chaining function of the parent chain needs to be invoked when a sub-chain ends (to continue the parent chain). Accordingly, when a host starts a sub-chain, the RPCC library saves the chaining function and its state parameter, and passes them along the sub-chain. The sub-chain ends when its chaining function returns null in `nexthop.server`, and a

result in `nexthop.state` (this is the result that the host originating the sub-chain must produce for the parent chain). When that happens, the RPCC library calls the saved chaining function with the saved state and `nexthop.state`.

It is important to note that a chain and a sub-chain are not necessarily aware of each other. For example, in Figure 5.4, the fact that the service function at B launches a sub-chain has little to do with the parent chain that called the service at B.

To allow multiple levels of nesting, we use a *chain stack* that stores the saved chaining function and its state for each level of composition. The stack is popped as each sub-chain ends.

5.2.5 Support for legacy RPC services

To support legacy services with standard RPC interfaces, we use a simple wrapper module, installed at the legacy RPC server, which includes the RPCC library and exposes the legacy remote procedures as service functions.

Each service function passes requests and responses to and from the corresponding legacy remote procedure. Because the service function calls the legacy remote procedure locally through the RPC's standard network interface (e.g., TCP), the legacy server will see all requests as coming from the local machine, and this can affect network address-based server access control policies. (This is not a problem if access control is based on internal RPC authenticators, such as signatures or tokens, which can be passed on by the wrapper.)

One solution is to re-implement the access control mechanism at the wrap-

per, but this is application-specific. A better solution is for the wrapper to fake the network address of its requests and capture the remote procedure's output before it is placed on the network.

5.2.6 Isolation

Chaining functions are pieces of client code running at servers. Even though clients are trustworthy in the environment we consider, they are still prone to buffer overruns, crashes, and other problems. Thus, chaining functions are sandboxed to provide isolation, so that client code cannot crash or otherwise adversely affect the server on which it runs.

We need two types of isolation: (1) restricting access to sensitive functions, such as file and network I/O and privileged operating system calls, and (2) restricting excessive consumption of resources (CPU and memory).

We achieve (1) through direct support by .NET/C# of access restrictions to file I/O, system and environment variables, registry, clipboard, sockets, and other sensitive functions (Java has similar capabilities). This is accomplished by placing descriptive annotations, called *attributes*, in the source code of chaining functions when they are compiled at run-time.

We achieve (2) by monitoring CPU and memory utilization and checking that they are within preset values. The appropriate values are a matter of policy at the server, but for the short-lived type of executions that we target with RPC chains, chaining functions should consume at most a few CPU seconds and hundreds of megabytes of memory, even in the most extreme cases.

If a chaining function violates restrictions on access or resource consumption, an RPC chain exception is thrown according to the mechanism in Section 5.2.8.

5.2.7 Debugging and profiling

A very useful debugging tool for traditional applications is “printf”, which allows an application to display messages on the console. We provide an analogous facility for RPC chain applications: a *virtual console*, where nodes in the chain can log debugging information. The contents of the virtual console are sent with the chain, and eventually reach the client, which can then dump the contents to a real console or file. The virtual console can also be used to gather profiling information for each step in the chain and be aggregated at the client.

Even with “printf”, debugging RPC chains can be hard, because it involves distributed execution over multiple machines. We can reduce this problem to the simpler problem of debugging RPC-based code by running RPC chains in a special *interactive mode*. The key observation is that chaining functions are *portable code* that can be executed at any machine. In interactive mode, chaining functions always execute at the client instead of the servers. To accomplish this, after each service function returns, the RPCC library sends its result back to the client, which then applies the chaining function to continue the chain from there. A chain executed in interactive mode looks like a series of RPC calls. By running the client in an interactive debugger, the developer can control the execution of the chain and inspect the outputs of service and chaining functions at each step.

5.2.8 Exceptions

An RPC chain may encounter exceptional conditions while it is executing: (1) the next server in the chain can be down, (2) the chaining function repository can be down, or (3) the state passed to the chaining function can be missing vital information due to a bug. All of these will result in an exception, either at the RPCC library in cases (1) and (2), or at a chaining function in case (3). (Service functions do not throw exceptions; they simply return an error to the caller.)

Who should handle such exceptions? One possibility is to handle them locally, by having the client send exception handling code as part of the chain. Doing this requires sending all the state that the handling code needs, which complicates the application design. Instead, we choose a less efficient but simpler alternative (since exceptions are the rare case). We simply propagate exceptions back to the client that started the chain. The client receives the exception name, its parameters, and the path of hosts that the chain has traversed thus far. (If the client crashes, the exception becomes moot and is ignored.)

In the case of nested chains, the exception propagates first to the host that started the current sub-chain. If that host does not catch the exception, it continues propagating to the host that started the parent chain, until it gets to the client. For example, in Figure 5.4 right, if E throws an exception (say, because it could not contact F), the exception goes to B, the node that created the sub-chain. This is a natural choice because B understands the logic of the sub-chain that it created, and so it may know how to recover from the exception. If B does not catch the exception, it is propagated to A.

5.2.9 Broken chains

The crash of a host while it executes an RPC chain results in a *broken chain*. These broken chains must be detected and handled gracefully.

Detection. We detect a broken chain using a simple end-to-end timeout mechanism at the client called *chain heartbeats*: a chain periodically sends an alive message to the client that created it, say every 3 seconds, and the client uses a conservative timeout of 6 seconds. If there are sub-chains, only the top-level creator gets the heartbeats. Heartbeats carry a unique chain identifier, a pair consisting of the client name and a timestamp, so that the client knows to which chain it refers.

We achieve the periodic sending through a time-to-heartbeat timer, which is sent with the chain, and it is decremented by each node according to its processing time, until it reaches 0, the time to send a heartbeat. Synchronized clocks are not needed to decrement the timer; we only need clocks that run at approximately the same speed as real time. Since we do not know link delays, we assume a conservative value of 200 ms and decrement the time-to-heartbeat timer by this amount for every network hop. This assumption may be violated when if there is congestion and dropped packets, resulting in a premature timeout (false positive). However, the impact of false positives is small because of our recovery mechanism, explained next.

Recovery. To recover from a broken chain, the client simply retransmits the request. Like standard remote procedures, we make chains idempotent by including a chain-id with each chain, and briefly caching the results of service functions and chaining functions at each server. If a server sees the same chain-

id, it uses the cached results for the service and chaining functions. The chain can continue in this fashion up to the host where the chain previously broke. At that host, if the “next” host is still down, an exception is thrown. Alternatively, a fail-over mechanism that calls a backup server can be implemented by using logical server names which are mapped to a backup when the primary fails. This is similar to the mechanisms used to fail over standard RPC’s.

Upon a second timeout, a client executes the RPC chain in *interactive mode* (as in Section 5.2.7), to determine exactly at which node the chain stopped, and returns an error to the application.

5.2.10 Splitting chains

For performance reasons, it may be desirable to split a chain to allow parallel execution. The decision to split a chain should be made with consideration of the added complexity, as concurrent computations are always harder to understand, design, debug, and maintain compared to sequential computations. Although our applications do not use splitting chains, we now explain how such chains can be implemented.

Split. We modify chaining functions so that they can return more than one *nexthop* parameter. The RPCC library calls each nexthop concurrently, resulting in the several split-chains. Each chain has an id comprised of the id of the parent plus a counter. For example, if there is a 3-way split of chain 74, the split-chains will have ids 74.1, 74.2, and 74.3. Each of these split chains can in turn be split again, and result in split-chains with increasingly long ids. For example, if split-chain 74.1 splits into two, the resultant split-chains will have ids 74.1.1

and 74.1.2. We note for future reference that each split-chain knows how many siblings it has (this information is passed on to the split-chains when the chain splits).

Broken split chains. Recall that we use an end-to-end mechanism to handle broken chains (Section 5.2.9) via a chain heartbeat. When a chain splits, we also split the heartbeats: each split-chain sends its own heartbeat (with the split-chain id) and the client will be content only if it periodically sees the heartbeat from all the split-chains. The heartbeat messages indicate the number of sibling split-chains, so that the client knows how many to expect. If a split-chain is missing, the client starts the chain again (even if other split-chains are still running, this does not cause a problem because of idempotency).

Merge. To merge split-chains, a *merge host* collects the results of each split-chain and invokes a *merge function* to continue the chain. The merge host and function are chosen when the chain splits (they are returned by the chaining function causing the split). The merge host can be any host; a good choice is the next host in the chain. The merge host awaits outcomes from all split-chains before calling the merge function, which takes the vector of results and returns *next hop*, specifying the next service function and chaining function to call.

After split-chains complete (i.e., reach the merge host), the parent chain will continue and resume its heartbeats. However, split-chains do not necessarily complete at the same time, so there may be a period from when the first split-chain completes until the parent chain resumes. During this period the merge host sends heartbeats on behalf of the completed split-chains, so that the client does not time out.

Crash garbage. When there are crashes in the system, the merge host may end up with the outcome of stale split-chains. This garbage can be discarded after a timeout: as we mentioned, RPC chains are intended for short-lived computations, so we propose a timeout of a minute. Note that if a slow system causes a running chain to be garbage collected, the client will recover after it times out.

5.3 Applications

To demonstrate RPC chains, we apply and evaluate them in two important enterprise applications: a storage application (Section 5.3.1) and a web application (Section 5.3.2).

5.3.1 Storage applications

Storage services generally provide two basic functions, *read* and *write*, based on keys, file names, object id's, or other identifiers. While this generic interface is suitable for many applications, its low-level nature sometimes forces bad data access patterns on applications. For instance, if a client wants to copy a large object from one storage server to another, the client must read the object from one server and write it to the other, causing all the data to go through the client. If the client is separated from the storage servers by a high latency or low bandwidth connection, this copying could be very slow.

One solution is to modify the storage service on a case-by-case basis for different operations and different settings. For example, the Amazon S3 stor-

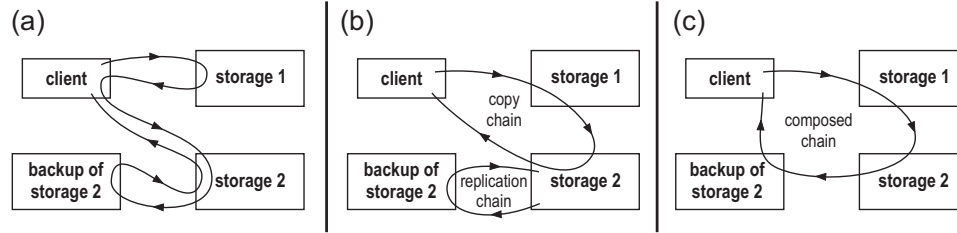


Figure 5.5: Copying data between two storage servers using RPCs, RPC chains, and RPC chains with composition.

age service recently added a new copy operation to its interface [4], so that an end user can efficiently copy her data between data centers in the US and Europe, without having to transfer data through her machine. Although such application-specific interfaces can be beneficial, they are specific to particular operations and do not mitigate adverse communication patterns in other settings.

RPC chains provide a more general solution: they not only enable the direct copying of data from one server to another (through a simple chain that reads and then writes), but also enable broader uses. To demonstrate this idea, we layered RPC chains over a legacy NFS v3 storage server, as explained in Section 5.2.5. (We could have used other types of storage, such as an object store.) We then implemented a simple chain to copy data without passing through the client.

We also show a more sophisticated application of chains by implementing a primary-backup replication of the storage server: when the primary receives a write request, it creates a chain to apply the request on a backup server. Because replication is done through chains, it can be composed with other chains. This is illustrated in Figure 5.5(b), which shows a setup with two storage servers,

the second of which is replicated, and a user who wants to copy data from the first to the second server. Two chains are created as a result of this request: a chain that the client launches for copying, and another that the second storage server launches for replication. The RPCC library allows these two chains to be composed together, as shown in Figure 5.5(c). We report on quantitative benefits of our approach in Section 5.4.3.

5.3.2 Web mail application

Web applications are generally composed of multiple tiers or services: there are front-end web servers, authentication servers, application servers, and storage and database servers. Some of these tiers, namely the web servers and application servers, play the role of orchestrating other tiers, and they tend to keep very little user state of their own, other than soft session state. This is a propitious setting for RPC chains, because performance gains can be realized by optimizing the communication patterns of the various services. We demonstrate this point with a sample application.

We consider a typical web mail application. There are web servers that handle HTTP requests, authentication servers and address-book servers that are shared with other applications, email storage servers that store the users' mail, and ad servers that are responsible for displaying relevant ads. These services can be located in multiple data centers, for several reasons: (1) no single data center can host them all; (2) a service may have been developed in a particular location and so it is hosted close by; (3) for performance reasons, it may be desirable for some services to be located close to their users (e.g., users created in

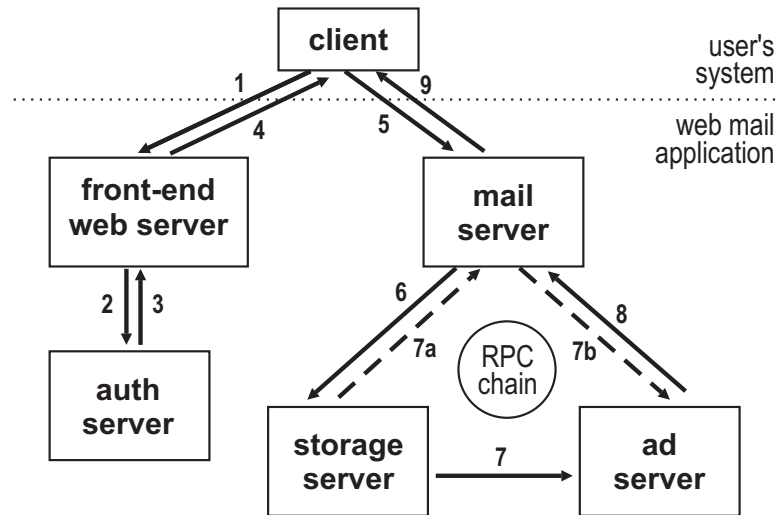


Figure 5.6: A simplified web mail server that uses RPC chains.

Asia may have their mailbox stored in Asia), though this is not always achievable (e.g., an Asian user travels to the U.S. and his mailbox is still in Asia); and (4) a service may need high availability or the ability to withstand disasters.

We implemented a simple web mail service as shown in Figure 5.6, to study the benefits of RPC chains in such a setting. Our web mail system consists of a front-end server that authenticates users by verifying their logins and passwords. Upon successful authentication, the front-end server returns a cookie to the client along with the name of an email server. The client then uses the cookie to communicate with the email server to send and receive email messages. Upon receiving a client request, the email server first verifies the cookie, then calls the back-end storage server to fetch the appropriate emails for the user. Finally, the mail server sends the message to an ad server so that relevant ads can be added to the messages before they are returned to the client.

Note that the adding of ads to emails imposes a significant overhead on performance. This is of particular concern because one of the primary performance

goals of a webmail service is to minimize the response time observed by clients. In addition, emails and ads cannot be fetched in parallel, since relevant ads cannot be selected without knowing the contents of the emails. It is also difficult to pre-compute the relevant ads because the relevance of ads may change over time.

Using RPC chains, we can mitigate some of the ad-related overheads. Even though we can only fetch ads after fetching the emails, we can eliminate one latency hop from the communication path of the web mail application by creating a chain that causes emails to be sent directly from the storage server to the ad server, without having to go through to email server (as shown in step 7 of Figure 5.6). Once the ad server has appended the appropriate ads to the emails, the emails can be sent to the email server which then returns it to the client. In Section 5.4.4, we evaluate the benefit of using RPC chains to improve the communication pattern in this fashion.

5.4 Evaluation

We now evaluate RPC chains. We start with some microbenchmarks, in which we measure the overhead of chaining functions and we compare RPC chains versus standard RPC's. We then evaluate the storage and web applications to demonstrate the performance improvements provided by RPC chains. The general question we address is when are RPC chains advantageous and what are the exact benefits.

		Redmond	Beijing	Cambridge
(a)	Mt.View	32 ms	180 ms	240 ms
	Redmond		146 ms	210 ms
	Beijing			354 ms
		Redmond	Beijing	Cambridge
(b)	Mt. View	6.3 MB/s	2.1 MB/s	1.4 MB/s
	Redmond		8.5 MB/s	8.6 MB/s
	Beijing			2.4 MB/s

Figure 5.7: Latency and bandwidth between pairs of sites.

5.4.1 Setup

Our experimental setup consists of ten machines in four geodistributed sites in a corporate network that spans the globe. We had machines in 4 sites: (1) Mountain View, California, USA, (2) Redmond, Washington, USA, (3) Cambridge, United Kingdom, and (4) Beijing, China. The measured latency and throughput of the links between these sites are shown in Figure 5.7.

5.4.2 Microbenchmarks

Overhead of chaining functions

In our first experiment, we evaluate the overhead imposed by chaining functions (pieces of client code) at servers. We considered chaining functions of three sizes, 621 bytes, 5 KB, and 50 KB, corresponding to small, medium, and large functions.

We first measured the time it takes to compile a function at run-time. The results are shown in the first two columns of Figure 5.8, averaged over 10 runs

Source size (KB)	Compile time (ms)	Compiled size (KB)
0.6	45.7 ± 0.3	0.4
5	47.1 ± 0.4	4.6
50	76.0 ± 0.3	15.9

Figure 5.8: Overhead for compiling chaining functions and storing compiled code.

(\pm refers to standard error). We used a 3 Ghz Intel Core 2 Duo processor running Windows Vista Enterprise SP1. The functions were written in C# and compiled using Microsoft Visual Studio 2008.

We also did a linear regression with a larger set of points (17 sizes, with 10 runs each) and found that the cost of compilation is 44.8 ms plus 1 ms for each 5000 bytes of source code. We see that there is a large initial compilation cost of tens of milliseconds, which we do not want to pay every time we call the server in a chain.

We measured the size of the compiled code, shown in the third column of Figure 5.8. We see that it is very small (we initially thought it would be large, but this is not the case). This allows the server to cache even tens of thousands of chaining functions in less than 50 MB, which justifies our choice of doing so.

RPC chain versus standard RPC

In our next experiment, we compare the latency of an RPC chain versus standard RPC. We used the smallest non-trivial chain, which goes through two servers (A chain that goes through only one server is the same as an RPC), and compare it against a pair of consecutive RPC's going to the two servers,

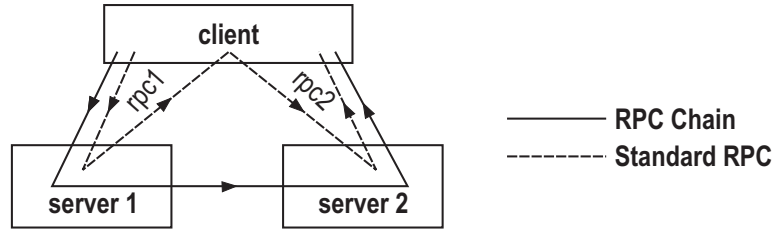


Figure 5.9: Executions used in the experiment of Section 5.4.2.

as shown in Figure 5.9. To isolate concerns, the service executed at each server is a no-op.

The figure makes it clear that the RPC chain incurs one fewer hop than the pair of RPC calls. What is not shown is that the RPC chain has potentially two overheads that the pair of RPC calls do not: (1) even if the client needs the response from server 1 but server 2 does not, the data is still relayed through server 2, and (2) the client needs to send state for the chaining function to execute at server 1. The first overhead can be avoided through a simple extension to RPC chains to allow each server in the chain to send some data to the client. i

We now consider the second overhead, and examine the question of how much state the client can send while still allowing the RPC chain to be faster than the pair of RPC calls. We assume that the chaining function is already cached at server 1, which is the common case for frequent chains.

Back-of-the-envelope calculation. We start with a simple calculation. Let S be the size of the state sent by the client for the chaining function at server 1. Then, in terms of total latency, the RPC chain saves one network latency but incurs $S/\text{link_bandwidth}$ to send the state. Thus, the RPC chain fares better as long as $\text{link_latency} > S/\text{link_bandwidth}$, or

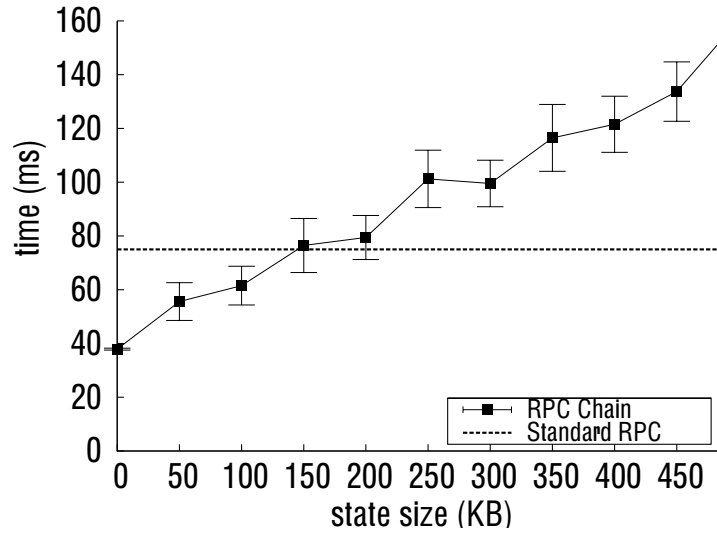


Figure 5.10: Execution time using an RPC chain versus standard RPC to call 2 servers.

$$S < link_latency \times link_bandwidth$$

For wide area links, the latency-bandwidth product can easily be in the tens to hundreds of kilobytes or more.

Experiment. We executed the RPC chain and the pair of RPC's. The client was located in Redmond while the servers were in Mountain View. (Because both servers were in the same site, this setup favors the RPC chain by an additional network latency; we later explain the case when the servers are far apart.)

Figure 5.10 shows the client end-to-end latency as a function of the state size (error bars show standard error). For the standard RPC execution, state size does not affect total latency, since this state simply stays at the client. The total latency was 75 ± 1 ms. For the RPC chain, the latency naturally increases with the state size. The point at which both lines cross is at ≈ 150 KB. This is a fair

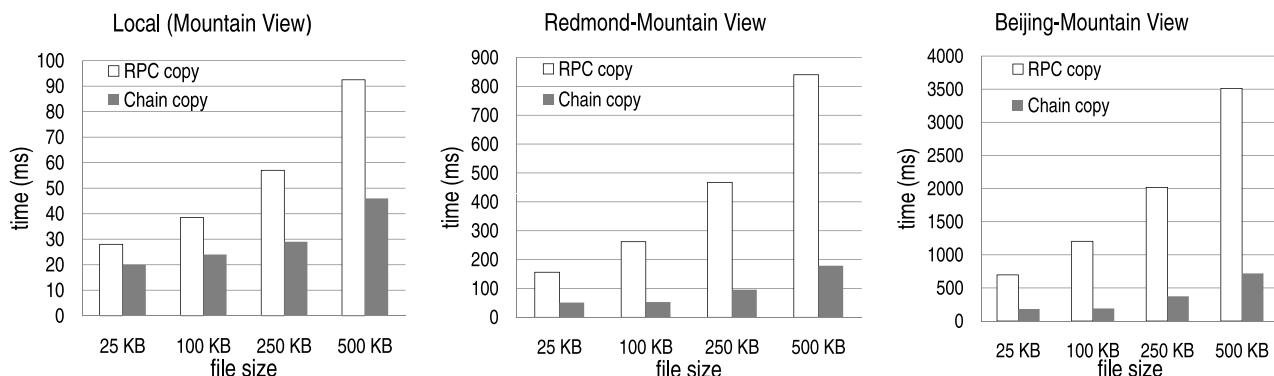


Figure 5.11: Comparison of RPC copy and Chain copy under various settings: clients collocated with servers in mountain view; client in redmond; and client in Beijing.

amount of state to send in many cases—definitely much more than we needed in either of our applications.

If servers 1 and 2 were far apart, this would shift the RPC chain line up by the corresponding extra latency. For example, if the latency from server 1 to server 2 were 15 ms, the lines would cross at ≈ 100 KB (assuming the distance from client to server 2 remains the same), which is still a reasonable state size (and much more than we needed in our applications).

5.4.3 Storage application

We now evaluate the use of RPC chains for the storage application described in Section 5.3.1.

Copy performance. In our experiments, we copy data from one storage server to another using two utilities: one that uses RPC chains, called *Chain copy*, and another that uses standard RPC's, called *RPC copy*. Both utilities use pipelining, so that the client has multiple outstanding requests on either server.

We also tried using the operating-system provided “copy” program, but it performed much worse than either Chain copy or RPC copy, because it reads and writes one chunk of data at a time (no pipelining).

In our first experiment, a single client copies a file of variable size (25 KB, 100 KB, 250 KB, and 500 KB) between two servers, and we measure the time it takes. We vary the location of the client (Mt. View, Redmond, Beijing) and fix the location of the servers in Mt. View. In the setting where both the client and the servers were in Mt. View, we placed them in two separate subnets, where the ping latency between the two was 2 ms and TCP bandwidth was 10 MB/s.

Figure 5.11 shows the results. Each bar represents the median of 40 repetitions of the experiment. As we can see, Chain copy provide considerable benefits in every case, compared to RPC copy. The benefits are greater for larger files and longer distances between client and servers. In a local setting, the copying time is reduced by up to factor of 2, while in the longest-distance setting (Beijing-Mt. View), the reduction is up to a factor of 5.

Another benefit of using Chain copy (not shown) is a reduction by a factor of two in (a) the aggregate network bandwidth consumption, and (b) the client bandwidth consumption. This reduction is important because links connecting data centers have limited bandwidth and/or are priced based on the bandwidth used.

In our next experiment, we vary the number of clients simultaneously copying files from one server to another, and measure the resultant throughput and latency of the system. This allows us to observe the behavior of the system under varying load as well as measure the peak throughput of the system. As

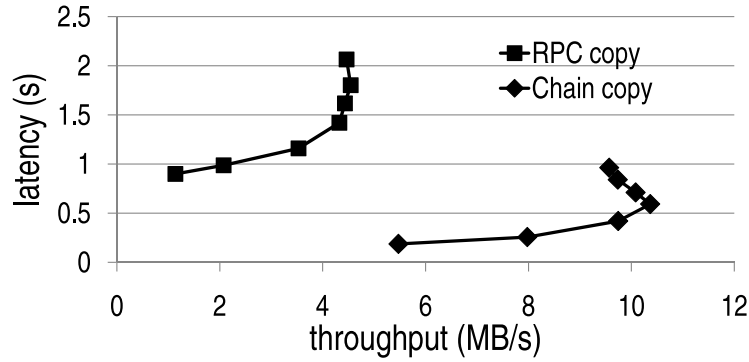


Figure 5.12: Throughput-latency of RPC copy and Chain copy.

before, the client machine was located in Redmond and the servers were located in Mt. View. We ran multiple client instances in parallel on the client machine, each client copying 1000 files in succession, each file measuring 256 KB in size. We measure the time that each client takes to complete copying 1000 files, and compute conservative throughput and latency numbers based on the slowest client.

Figure 5.12 shows the results of the experiment. For both RPC copy and Chain copy, the average latency decreases as the amount of workload placed on the system increases. Initially, the increase in workload also results in an increase in the aggregate throughput of the system, but once the system becomes saturated, any increase in workload only increases latency without any gain in throughput. Our results show that RPC copy is able to sustain a peak throughput of 4.5 MB/s. This peak throughput occurs when the network link between the client and the servers, which had a bandwidth of 6.3 MB/s, becomes saturated. Since Chain copy does not require that the data blocks of the files being copied actually flow through the client, it was not subject to this limitation and was thus able to achieve a higher peak throughput of 10.4 MB/s. Rather

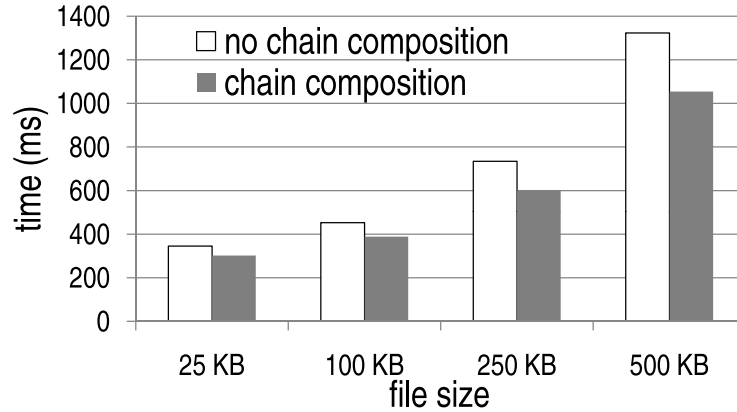


Figure 5.13: Benefit of chain composition.

than a network bandwidth limitation, Chain copy's throughput is limited by the servers' ability to keep up with requests.

Benefit of chain composition. In this experiment, we measure the benefit of composing RPC chains. We use two chains: one for copying from one server to another (as above) and the other for primary-backup replication of the second server (as in Figure 5.5). We compare two systems that use RPC chains; one system uses chain composition to combine the two chains, while the other has composition disabled. In the experiment, one client copies one file of variable size from the non-replicated server to the replicated server. The client is in Cambridge, the source server is in Mt. View, the primary of the destination server is in Mt. View, and the backup of the destination server is in Redmond.

Figure 5.13 shows the result. As we can see, composing the chain reduces the duration of the copy by 12%-20%, with larger files having a greater reduction. Without composition, the destination server has to handle both requests from the source server as well as the replies from the backup server. Composition reduces the load on the destination server by allowing the backup server

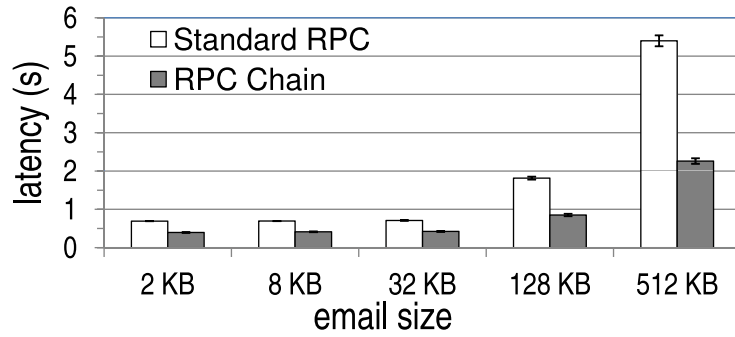


Figure 5.14: RPC chain in web mail application.

to send replies directly to the client. In addition, composition eliminates the unnecessary messages from the backup server to the destination server, reducing the amount of bandwidth consumption. A combination of these factors allow composition to improve the overall performance of the system. As file size increases, the setup cost becomes relatively small compared to the actual cost of executing the chains. This makes the impact of the more efficient chain that resulted from composition more apparent.

5.4.4 Web mail application

We now describe the evaluation of the web mail application presented in Section 5.3.2. In our experimental setup, we placed the client in Mountain View, the mail server and the authentication server in Redmond, and all other servers in Beijing. This setup emulates the case where a user from Asia travels to the US and wants to access web mail services that are hosted in Asia. Since the web mail provider may have servers deployed worldwide, the user can be directed to a mail server and an authentication server (Redmond) that is close to his current location (Mountain View). However, user-specific data is stored on servers

close to the user's normal location (Beijing), so the mail server has to fetch data from those machines.

Specifically, after receiving a cookie from the client and verifying the client's identity, the mail server must fetch the client's email from the storage server followed by appropriate ads from the ad server, both of which are located in Beijing. A traditional system implemented using RPC's would have the mail server contact the storage server, fetch the user's emails, then contact the ad server to retrieve relevant ads. However, in our setting, where the mail server is located close to the client but far away from the storage server and ad server, traversing the long links between Redmond and Beijing four times would be less than ideal. As described in Section 5.3.2, RPC chains allow us to eliminate unnecessary network traversals. In this case, our RPC-chain-enabled mail server sends emails directly from the storage server to the ad server before returning the result to the mail server, halving the number of long link traversals.

We measure the client perceived latency of opening an inbox and retrieving one email: the client first contacts the front end authentication server to authenticate herself, then she sends a read request to the mail server to retrieve a single email. We measure the time it takes for the client to receive the email, which is appended with an ad whose size is small relative to the size of the email. We vary the size of the email that is fetched, and for each size, we repeated the experiment 20 times.

As shown in Figure 5.14, RPC chains consistently reduces the client perceived latency of the web mail application. As the size of the email increases, the latency improvement from using RPC changes also increases. Overall, we found that the use of RPC chains reduced the latency of the web mail applica-

tion by 40% to 58% when compared to standard RPC's.

We note that the significant performance gains of using RPC chains comes at a very low cost of implementation. For the web mail application, the effort involved in enabling RPC chains was mainly in terms of implementing chaining functions which totaled a mere 48 lines of C# code. In general, a simple way for existing applications to benefit from RPC chains is to identify the critical causal path of RPC requests, and replace that path with an RPC chain. The effort is that of writing a single RPC chain; in the worst case, one can do it from scratch. The harder problem is finding the critical causal path, which has been an active area of research (e.g., [3]).

5.5 Limitations

Chaining state cannot always be sent. RPC chains are not appropriate if the chaining state is large or if it cannot be determined when the client starts the chain. For example, suppose that (1) A calls B using an RPC, (2) A gets a reply, and (3) depending on the state of a sensor or some immediate measurement at A, A then calls C or D. It is not possible to use an RPC chain $A \rightarrow B \rightarrow (C \text{ or } D)$, because the choice of going to C versus D must be made at A where the sensor is.

Programming with continuations. To use RPC chains, developers need to make use of continuation-style programming. This can be much harder than programming using sequential code, because continuations must explicitly keep track of all their state. Continuations are notoriously hard to debug, because there is no simple way to track the execution that led to a given state.

We note, however, that programming with continuations is already tolerated in code that uses asynchronous RPC's and callbacks. Moreover, one could perhaps write a tool that automatically produces continuations from sequential code, using techniques from the compiler literature (see, e.g., [5]).

Terminating chains. When an application terminates, it is usually desirable to release its resources and halt all its activities. However, if the application has outstanding RPC chains, it is not easy to terminate them. This problem exists with traditional RPC's as well (there is no easy way to terminate a remote procedure), but it is worse with RPC chains because the remote servers involved may not be known.

RPC chains are designed for relatively short-lived executions, and for these uses, this problem is less of a concern, because a chain soon terminates anyways. The only exception is a buggy chain that runs forever. For such chains, the RPCC library can impose a maximum chain length, say 2000 hops, and throw an exception after that.

CHAPTER 6

CONCLUSION AND FUTURE WORK

Distributed systems are here to stay. As more and more of the systems that developers build become distributed, it is important to provide developers with the tools to make it easy to build robust, high-performance distributed applications. In this thesis, we have looked at several different ways towards realizing this goal.

In Chapter 2, we demonstrated that while many well-known consensus protocols such as Paxos, Chandra-Toueg, and Ben-Or at first appear different, they share an underlying commonality. We distilled this commonality into the form of a single skeleton algorithm that can be instantiated into each of these specific protocols by configuring the quorum systems that are used, the way instances are started, and other protocol-specific details. Using this approach, we implemented the skeleton algorithm and used it to instantiate Ben-Or, Chandra-Toueg, and two variants of the Paxos algorithm. Simulation experiments using our implementation allowed us to explore the effect of the differences between these algorithms on their performance under different workloads and crash failures. We believe that the skeleton algorithm is an interesting basis for the understanding of consensus algorithms and comparison of their performance, and provides a novel platform for the exploration of other possible consensus protocols.

We demonstrated that well-known consensus protocols such as Paxos, Chandra-Toueg, and Ben-Or can be derived from a single skeleton. An interesting feature of this skeleton is that it uses not one but two quorum systems. By instantiating these quorum systems, as well as some other details of the skele-

ton, specific consensus protocols can be derived. We have implemented this skeleton and have used it as the basis for a performance comparison.

In Chapter 3, we examined a potential way to reduce the performance cost of building Byzantine fault tolerant systems. While Byzantine fault tolerance aims to provide resilience against arbitrary failures, in many applications, failures and contention are not the norm. Here, we have examined optimization opportunities in contention-free and failure-free situations. We provided two definitions of one-step asynchronous algorithms that provide low latency performance in favorable conditions while guaranteeing strong consistency when failures and contention occur. We proved lower bounds in the number of processors required for such algorithms, and presented Bosco, an algorithm that meets those bounds.

In Chapter 4, we generalize our approach and discuss one-step algorithms in the context of k -set agreement. We presented an algorithm that solves this problem for crash faults when $n > fk - k^2 + f + \max(2k, f + 1)$ and showed that this is optimal for the class of popularity-based protocols.

The number of processors that the algorithm requires is modest for small values of k . For $k = 1$, the consensus problem, our algorithm requires $3f + 1$ processors, which matches the lower bound for one step agreement [13]. For $k = 2$, the algorithm requires $4f - 2$ processors. It is also a modest requirement for values of k near f , requiring $3f + 1$ and $4f - 2$ for $k = f$ and $k = f - 1$ respectively. For intermediate values of k , the number of nodes required can be quite large. For $k = f/2$, it requires $f^2/4 + 2f + 2$ processors. So if there is room for significant improvement on this problem, it would be in this range.

One noteworthy feature of our algorithm is that it provides a very simple but non-trivial way to turn a consensus algorithm into a k -set agreement algorithm. Since implementing new distributed computing algorithms is generally time consuming and error-prone, this allows an implementer to re-use an existing implementation and, if contention is relatively rare, gain almost the full benefit of a more general k -set agreement algorithm.

We have not proven a general lower bound for this problem, but Lemma 32 shows that any correct algorithm can be broken into a decision rule for what value to decide if at most k values are received in the first round and an algorithm for reaching a decision in the general case. Our algorithm uses a natural decision rule: decide on the most popular value (using an ordering over values to break ties). Theorem 35 shows that our algorithm is optimal among algorithms that use this decision rule. This implies that attempts to find better algorithms must focus on alternative decision rules.

Obvious alternative decision rules we have examined do not seem promising. One option is to sometimes decide on a less popular value. However, this requires a processor to decide on a value that, should other processors not learn of its decision, has less support for them deciding on it. Another option is to bias either towards or away from a processor's local value (for example when breaking ties). We have been unable to come up with an algorithm of this flavor that performed well. As such, finding a true lower bound or an improved algorithm remains an open problem.

Finally, in Chapter 5, we proposed the RPC chain, a simple but powerful primitive that combines multiple RPC invocations into a chain, in order to optimize the communication pattern of applications that use many composite ser-

vices, possibly developed independently of each other. With RPC chains, client can save network hops, resulting in considerably smaller end-to-end latencies in a geodistributed setting. Clients can also save bandwidth because they are not forced to receive data they do not need. We demonstrated the efficacy of RPC chains for a storage and a web application, and believe that RPC chains can be applied in many other applications.

While our implementation of RPC chains proved effective in our sample applications, we believe that there are several useful extensions that can be built into RPC chains. One possible extension is the support of intermediate chain results. If a client wants to receive some results from intermediate servers of the chain, these results need to be relayed through the chain. If the amount of data is large, it can impose a significant overhead. We can extend RPC chains to address this issue, by allowing each server in the chain to directly return some data to the client.

Another useful extension to RPC chains is the support for large chaining states. The chaining state is the state that the client sends along the chain to execute the chaining functions. If this state is large, this can incur a significant state overhead. Two optimizations are possible to mitigate this cost.

Fall-back to standard RPC. As explained in Section 5.2.7, we can execute a chain in interactive mode, which causes the chain to go back to the client at every step. This is effectively a fall-back to standard RPC, causing all chaining functions to execute at the client, which eliminates the overhead of sending the chaining state, at the cost of extra network delays. We explored this trade-off in Section 5.4.2. It is possible to have the RPCC library gauge the size of the chaining state before starting the chain, and if the state is larger than some threshold,

execute the chain in interactive mode. The threshold can be chosen dynamically based on previous executions of the same chain, in an adaptive manner. By doing so, an RPC chain will always perform at least as well as standard RPC's, modulo the small computational overhead of executing chaining functions and the time it takes to adapt.

Hiding latency. In our implementation, servers wait to receive the chaining state before executing the next service function in the chain. This waiting is not necessary, because the service function depends only on its parameters, not on the chaining state (the chaining state is only needed for the chaining function, which executes later). Therefore, a natural optimization is to start the service function even as the chaining state is being received. If the service function takes significant time to complete, (e.g., it involves disk I/O or some lengthy computation), this will mask part or all of the latency of transmitting the chaining state.

In conclusion, while much work remains to be done, we believe that our contributions in this thesis moves us closer to the goal of providing developers with the tools to make the development of distributed applications as painless as possible.

BIBLIOGRAPHY

- [1] Apache zookeeper. <http://hadoop.apache.org/zookeeper/>.
- [2] Michael Abd-El-Malek, Gregory R. Ganger, Garth R. Goodson, Michael K. Reiter, and Jay J. Wylie. Fault-scalable Byzantine fault-tolerant services. *SIGOPS Operating Systems Review*, 39(5):59–74, 2005.
- [3] Marcos K. Aguilera, Jeffrey C. Mogul, Janet Wiener, Patrick Reynolds, and Athicha Muthitacharoen. Performance debugging for distributed systems of black boxes. In *ACM Symposium on Operating Systems Principles*, pages 74–89, October 2003.
- [4] Amazon.com, Inc. Amazon simple storage service: Copy proposal. <http://doc.s3.amazonaws.com/proposals/copy.html>.
- [5] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.
- [6] Yariv Aridor and Mitsuru Oshima. Infrastructure for mobile agents: Requirements and design. In *Workshop on Mobile Agents*, pages 38–49, September 1998.
- [7] D. Barbará, S. Mehrotra, and M. Rusinkiewicz. INCAs: Managing dynamic workflows in distributed environments. *Journal of Database Management, Special Issues on Multidatabases*, 7(1):5–15, Winter 1996.
- [8] M. Barborak and M. Malek. The consensus problem in fault-tolerant computing. *ACM Computing Surveys*, 25(2), 1993.
- [9] M. Ben-Or. Another advantage of free choice: Completely asynchronous agreement protocols. In *Proc. of the 2nd ACM Symp. on Principles of Distributed Computing*, pages 27–30, Montreal, Quebec, August 1983. ACM SIGOPS-SIGACT.
- [10] Martin Biely, Josef Widder, Bernadette Charron-Bost, Antoine Gaillard, Martin Hüttele, and André Schiper. Tolerating corrupted communication. In *PODC '07: Proceedings of the twenty-sixth annual ACM symposium on Principles of Distributed Computing*, pages 244–253, New York, NY, 2007. ACM.
- [11] Romain Boichat, Partha Dutta, Svend Frolund, and Rachid Guerraoui. Reconstructing Paxos. *ACM SIGACT News*, 34, 2003.

- [12] Elizabeth Borowsky and Eli Gafni. Generalized flp impossibility result for t-resilient asynchronous computations. In *Proc. of the 25th Annual ACM Symposium on Theory of Computing*, pages 91–100, San Diego, California, 1993.
- [13] Francisco V. Brasileiro, Fabíola Greve, Achour Mostéfaoui, and Michel Raynal. Consensus in one communication step. In *Proc. of the 6th International Conference on Parallel Computing Technologies*, pages 42–50, London, UK, 2001. Springer-Verlag.
- [14] Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, pages 335–350, Berkeley, CA, USA, 2006. USENIX Association.
- [15] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *Proc. of the 3rd Symposium on Operating Systems Design and Implementation (OSDI)*, New Orleans, LA, February 1999.
- [16] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. pages 173–186, 1999.
- [17] T.D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996.
- [18] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: an engineering perspective. In *PODC '07: Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 398–407, New York, NY, USA, 2007. ACM.
- [19] Bernadette Charron-Bost and André Schiper. The Heard-Of model: Unifying all benign failures. Technical Report LSR-REPORT-2006-004, EPFL, June 2006.
- [20] Soma Chaudhuri. Agreement is harder than consensus: Set consensus problems in totally asynchronous systems. In *Proc. of 9th Annual ACM Symposium on Principles of Distributed Computing*, pages 311–324, Quebec City, Quebec, Canada, 1990.
- [21] Soma Chaudhuri. More choices allow more faults: Set consensus problems in totally asynchronous systems. *Information and Computation*, 105:132–158, 1993.

- [22] James Cowling, Daniel Myers, Barbara Liskov, Rodrigo Rodrigues, and Liuba Shrira. HQ replication: a hybrid quorum protocol for Byzantine fault tolerance. In *OSDI '06: Proceedings of the 7th symposium on Operating Systems Design and Implementation*, pages 177–190, Berkeley, CA, 2006. USENIX Association.
- [23] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *ACM Symposium on Operating System Design and Implementation*, pages 137–150, December 2004.
- [24] Dan Dobre and Neeraj Suri. One-step consensus with zero-degradation. In *DSN '06: Proceedings of the International Conference on Dependable Systems and Networks*, pages 137–146, Washington, DC, 2006. IEEE Computer Society.
- [25] M.J. Fischer, N.A. Lynch, and M.S. Patterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, April 1985.
- [26] Michael J. Freedman, Karthik Lakshminarayanan, Sean Rhea, and Ion Stoica. Non-transitive connectivity and DHTs. In *Conference on Real, Large Distributed Systems*, pages 55–60, December 2005.
- [27] Roy Friedman, Achour Mostefaoui, and Michel Raynal. Simple and efficient oracle-based consensus protocols for asynchronous Byzantine systems. *IEEE Transactions on Dependable and Secure Computing*, 02(1):46–56, 2005.
- [28] Robert Gray, David Kotz, Saurab Nog, Daniela Rus, and George Cybenko. Mobile agents: The next generation in distributed computing. In *Aizu International Symposium on Parallel Algorithms and Architectures Synthesis*, pages 8–24, March 1997.
- [29] Rachid Guerraoui, Nikola Knezevic, Vivien Quema, and Marko Vukolic. The Next 700 BFT Protocols. In *Proceedings of the 5th ACM European conference on Computer systems*, 2010.
- [30] Rachid Guerraoui and Michel Raynal. The information structure of indulgent consensus. In *Proc. of 23rd IEEE International Conference on Distributed Computing Systems*, 2003.
- [31] Rachid Guerraoui and Michel Raynel. The alpha of indulgent consensus. *The Computer Journal*, 2006.

- [32] Colin G. Harrison, David M. Chess, and Aaron Kershenbaum. Mobile Agents: Are they a good idea? In *International Workshop on Mobile Object Systems*, pages 25–47, July 1996.
- [33] Naohiro Hayashibara, Péter Urbán, André Schiper, and Takuya Katayama. Performance comparison between the Paxos and Chandra-Toueg consensus algorithms. In *Proc. of International Arab Conference on Information Technology*, pages 526–533, Doha, Qatar, December 2002.
- [34] James Hendricks, Gregory R. Ganger, and Michael K. Reiter. Low-overhead Byzantine fault-tolerant storage. In *Proc. of twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, pages 73–86, New York, NY, 2007. ACM.
- [35] Maurice Herlihy and Nir Shavit. The asynchronous computability theorem for t-resilient tasks. In *Proc. of the 25th Annual ACM Symposium on Theory of Computing*, pages 111–120, San Diego, California, 1993.
- [36] M. Hurfin, A. Mostéfaoui, and M. Raynal. A versatile family of consensus protocols based on chandra-toueg’s unreliable failure detectors. *IEEE Trans. Comput.*, 51(4):395–408, 2002.
- [37] Larry Huston, Rahul Sukthankar, Rajiv Wickremesinghe, M. Satyanarayanan, Gregory R. Ganger, Erik Riedel, and Anastassia Ailamaki. Diamond: A Storage Architecture for Early Discard in Interactive Search. In *USENIX Conference on File and Storage Technologies*, pages 73–86, March 2004.
- [38] Suresh Jagannathan. Continuation-based transformations for coordination languages. *Theoretical Computer Science*, 240(1):117–146, June 2000.
- [39] Idit Keidar and Sergio Rajsbaum. On the cost of fault-tolerant consensus when there are no faults. *SIGACT News*, 32(2):45–63, 2001.
- [40] Jonathan Kirsch and Yair Amir. Paxos for system builders: an overview. In *LADIS ’08: Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware*, pages 1–6, New York, NY, USA, 2008. ACM.
- [41] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: speculative Byzantine fault tolerance. In *Proc. of twenty-first ACM SIGOPS symposium on Operating Systems Principles*, pages 45–58, New York, NY, 2007. ACM.

- [42] L. Lamport. The part-time parliament. *Trans. on Computer Systems*, 16(2):133–169, 1998.
- [43] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *Trans. on Programming Languages and Systems*, 4(3):382–401, July 1982.
- [44] Leslie Lamport. Lower bounds for asynchronous consensus. Technical Report MSR-TR-2004-72, Microsoft Research, July 2004.
- [45] Leslie Lamport. Fast Paxos. *Distributed Computing*, 19(2):79–103, 2006.
- [46] Harry C. Li, Allen Clement, Amitanand S. Aiyer, and Lorenzo Alvisi. The paxos register. In *SRDS '07: Proceedings of the 26th IEEE International Symposium on Reliable Distributed Systems*, pages 114–126, Washington, DC, USA, 2007. IEEE Computer Society.
- [47] D. Malkhi and M.K. Reiter. Byzantine Quorum Systems. *Distributed Computing*, 11:203–213, June 1998.
- [48] Dahlia Malkhi, Michael K. Reiter, Avishai Wool, and Rebecca N. Wright. Probabilistic quorum systems. *Information and Computation*, 170(2):184–206, 2001.
- [49] J-P. Martin and L. Alvisi. Fast Byzantine consensus. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 402–411, June 2005.
- [50] Jean-Philippe Martin and Lorenzo Alvisi. Fast byzantine consensus. *IEEE Transactions on Dependable and Secure Computing*, 3:202–215, 2006.
- [51] Michael G. Merideth and Michael K. Reiter. Probabilistic opaque quorum systems. In Andrzej Pelc, editor, *DISC*, volume 4731 of *Lecture Notes in Computer Science*, pages 403–419. Springer, 2007.
- [52] Zarko Milosevic, Martin Hutle, and Andr Schiper. Unifying Byzantine Consensus Algorithms with Weak Interactive Consistency. In *Proceedings of the 13th International Conference On Principles Of Distributed Systems (OPODIS)*, 2009.
- [53] Luc Moreau. The PCKS-machine: An abstract machine for sound evaluation of parallel functional programs with first-class continuations. In *European Symposium on Programming*, pages 424–438, April 1994.

- [54] Achour Mostéfaoui, Sergio Rajsbaum, and Michel Raynal. Conditions on input vectors for consensus solvability in asynchronous distributed systems. *J. ACM*, 50(6):922–954, 2003.
- [55] Achour Mostéfaoui, Sergio Rajsbaum, Michel Raynal, and Corentin Travers. The combined power of conditions and information on failures to solve asynchronous set agreement. *SIAM J. Comput.*, 38(4):1574–1601, 2008.
- [56] Achour Mostéfaoui and Michel Raynal. Solving consensus using chandra-toueg’s unreliable failure detectors: A general quorum-based approach. In *Proceedings of the 13th International Symposium on Distributed Computing*, pages 49–63, London, UK, 1999. Springer-Verlag.
- [57] Achour Mostéfaoui and Michel Raynal. Solving consensus using Chandra-Toueg’s unreliable failure detectors: A general quorum-based approach. In *Proc. of the International Symposium on Distributed Computing*, pages 49–63, 1999.
- [58] Achour Mostefaoui and Michel Raynal. Randomized k-set agreement. In *Proc. of 13th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 291–297, Crete Island, Greece, 2001.
- [59] M. Naor and A. Wool. The load, capacity, and availability of quorum systems. *SIAM Journal on Computing*, 27(2):423–447, April 1998.
- [60] Jakob Nielsen. *Designing Web Usability: The Practice of Simplicity*. New Riders Publishing, Indianapolis, 1999.
- [61] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27(2):228–234, April 1980.
- [62] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard M. Karp, and Scott Shenker. A scalable content-addressable network. In *Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 161–172, August 2001.
- [63] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *International Conference on Distributed Systems Platforms*, pages 329–350, November 2001.

- [64] F.B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.
- [65] Yee Jiun Song, Marcos K. Aguilera, Ramakrishna Kotla, and Dahlia Malkhi. Rpc chains: efficient client-server communication in geodistributed systems. In *NSDI'09: Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, pages 277–290, Berkeley, CA, USA, 2009. USENIX Association.
- [66] Yee Jiun Song and Robbert van Renesse. Bosco: One-step byzantine asynchronous consensus. In *DISC*, pages 438–450, 2008.
- [67] Yee Jiun Song, Robbert van Renesse, Fred B. Schneider, and Danny Dolev. The building blocks of consensus. In *ICDCN*, pages 54–72, 2008.
- [68] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 149–160, August 2001.
- [69] Christopher Strachey and Christopher P. Wadsworth. Continuations: A mathematical semantics for handling full jumps. *Higher-Order and Symbolic Computation*, 13(1-2):135–152, April 2000.
- [70] David L. Tennenhouse, Jonathan M. Smith, W. David Sincoskie, David J. Wetherall, and Gary J. Minden. A survey of active network research. *IEEE Communications Magazine*, 35(1):80–86, January 1997.
- [71] Péter Urbán and André Schiper. Comparing distributed consensus algorithms. In *Proc. of International Conference on Applied Simulation and Modelling*, pages 474–480, 2004.
- [72] Jim White. Telescript technology: The foundation for the electronic marketplace, 1994. Unpublished manuscript. White paper, General Magic, Inc.
- [73] Jim White. Mobile agents white paper, 1996. Unpublished manuscript. Available at <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.55.7931>.
- [74] World Wide Web Consortium. SOAP version 1.2. <http://www.w3.org>.

- [75] Weihai Yu and Jie Yang. Continuation-passing enactment of distributed recoverable workflows. In *ACM Symposium on Applied Computing*, pages 475–481, March 2007.
- [76] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *ACM Symposium on Operating System Design and Implementation*, pages 1–14, December 2008.
- [77] Ben Y. Zhao, Ling Huang, Jeremy Stribling, Sean C. Rhea, Anthony D. Joseph, and John D. Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications*, 22(1):41–53, January 2004.
- [78] Piotr Zielinski. Optimistically terminating consensus: All asynchronous consensus protocols in one framework. In *ISPD '06: Proceedings of the Proceedings of The Fifth International Symposium on Parallel and Distributed Computing*, pages 24–33, Washington, DC, USA, 2006. IEEE Computer Society.
- [79] Piotr Zielinski. Automatic verification and discovery of byzantine consensus protocols. In *DSN '07: Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 72–81, Washington, DC, USA, 2007. IEEE Computer Society.